



Design Patterns for Ethereum Smart Contracts

TIAGO AUGUSTO PINTO MOURA

Julho de 2020

Design Patterns for Ethereum Smart Contracts

Tiago Moura

**A dissertation submitted in fulfillment of
the requirements for the degree of Master in Informatics,
Specialization Area of Information and Knowledge Systems**

Supervisor: Isabel de Fátima Silva Azevedo

Porto, July 5, 2020

Dedictory

Dedicated to my family and friends who have always encouraged me, giving me the strength to finish this dissertation.

Abstract

The Ethereum platform has become very popular in recent years in smart contract development, an area in which several programming languages are applied. Solidity, nevertheless is the most popular and widely used. On top of blockchain technology, Ethereum and its smart contracts have enabled developers worldwide to create innovative solutions in various areas such as finance, healthcare, insurance, internet of things, supply chain and others. These solutions are referred to as Decentralized Applications (dapps) and use the distributed nature of blockchain technology to solve problems in a disruptive manner, casting out intermediaries and automating processes. The fact that blockchain transactions triggered by smart contract execution are unchangeable also provides dapps users with a lot of confidence when compared to traditional centralized apps, because dapps' data becomes tamper-proof.

In the context of this work, it was necessary to analyze and look for software design patterns in order to help making the development of Solidity smart contracts simpler and safer, thus giving confidence both to users and developers of dapps built with Ethereum. In an initial phase, a literature review was made and 62 design patterns were found. From this set, 30 design patterns were selected for the practical component of this work. In this stage, 318 smart contracts were analyzed and the 30 previously selected patterns were confirmed. Additionally, four more patterns which had not been identified in the literature review stage were found.

The 30 design patterns were divided into four categories: authorization, control, maintenance and security. Efficiency across all these categories was evaluated in terms of gas consumption and security vulnerabilities. Once the research was completed, it became possible to create a set of recommendations for Solidity developers.

Keywords: Blockchain, Ethereum, Smart Contracts, Solidity, Security, Gas, Design Patterns

Resumo

A plataforma Ethereum tornou-se muito popular nos últimos anos no desenvolvimento de contratos inteligentes (smart contracts), uma área na qual várias linguagens de programação são aplicadas. Solidity, no entanto, é a mais popular e amplamente utilizada. Alicerçada na tecnologia blockchain, a Ethereum e os seus contratos inteligentes permitiram que programadores por todo o mundo criassem soluções inovadoras em várias áreas como finanças, saúde, seguros, internet das coisas, cadeia de abastecimento e outras. Essas soluções são chamadas de aplicações descentralizadas (dapps) e usam a natureza distribuída da tecnologia blockchain para resolver problemas de maneira disruptiva, eliminando intermediários e automatizando processos. O facto das transações registadas na blockchain, que são desencadeadas pela execução de cláusulas nos contratos inteligentes, também serem imutáveis dá aos utilizadores das dapps muita confiança em comparação com as aplicações centralizadas tradicionais, porque assim elas tornam-se à prova de adulteração de dados.

No contexto deste trabalho, foi necessário analisar e procurar padrões de desenho (design patterns) de software, que ajudassem a tornar o desenvolvimento de contratos inteligentes em Solidity mais simples e seguros, dando confiança tanto aos utilizadores como aos programadores nas dapps construídas através da Ethereum. Numa fase inicial, foi realizada uma revisão da literatura onde foram encontrados sessenta e dois padrões de desenho, dos quais trinta foram seleccionados para a componente prática deste trabalho. Durante esta componente, foram analisados trezentos e dezoito contratos inteligentes, onde os trinta padrões previamente seleccionados foram confirmados. Além disso, foram encontrados mais quatro padrões que não foram encontrados na revisão da literatura.

Os trinta padrões de design escolhidos foram divididos em quatro categorias: autorização, controlo, manutenção e segurança. Todas estas categorias foram avaliadas em termos de eficiência no consumo de gás e vulnerabilidades de segurança. Através de todo o trabalho realizado, tanto teórico quanto prático, foi possível criar um conjunto de recomendações para programadores de Solidity.

Acknowledgements

I would like to express my thanks to all those who have contributed to the development of this work.

Firstly to my supervisor **Isabel Azevedo**, for the support and availability provided throughout the realization of this work.

Secondly, to all my co-workers who have always encouraged me to finish my master's degree.

Finally, I would like to thank my family and friends for their support and patience.

Contents

List of Figures	xvii
List of Tables	xix
List of Source Code	xxii
List of Acronyms	xxiii
1 Introduction	1
1.1 Context	1
1.2 Problem	2
1.3 Objectives and Research Methodology	3
1.4 Document Structure	4
2 Background	5
2.1 Blockchain	5
2.1.1 Bitcoin	7
2.1.2 Consensus Mechanism	8
2.1.3 Blockchain Oriented Software Engineering	9
2.2 Ethereum	9
2.2.1 Ethereum Virtual Machine	10
2.2.2 Accounts	10
2.2.3 Fees	11
2.2.4 Smart Contracts	12
2.2.5 Solidity	13
Variables	14
Method visibility	15
2.2.6 Tools	15
Blockchain Explorer	15
Security	17
2.2.7 Alternative Platforms	17
EOS	17
NEO	18
Stellar	18
Cardano	18
Ethereum Alternatives Comparison	19
3 State of the Art	21
3.1 Literature Research	21
3.2 Authorization Patterns	30
3.3 Security Patterns	30
3.4 Maintenance Patterns	31

3.5	Control Patterns	31
3.6	Gas Economic Patterns	32
4	Value Analysis	33
4.1	Business and Innovation Process	33
4.2	Value, Customer Value and Perceived Value	34
4.2.1	Customer Value	35
4.2.2	Perceived Value	35
4.2.3	Benefits and Sacrifices	36
4.3	Current Market Valuation	36
4.3.1	Bitcoin DeFi Ecosystem	36
4.3.2	Ethereum DeFi Ecosystem	37
4.3.3	Market Valuation	37
4.4	The Analytic Hierarchy Process	38
5	Problem Statement	43
5.1	Main Issues	43
5.2	Objectives	45
6	Design and Implementation	47
6.1	Design	47
6.1.1	Selection of Technologies and Patterns	47
6.1.2	Detailed Approach	48
6.2	Collecting Smart Contracts	48
6.2.1	Text Mining Search Approach	50
6.2.2	Manual Search Approach	52
	Known Design Patterns Found	52
	Unknown Design Patterns Found	54
	Smart Contracts Taxonomy	55
6.3	Security Analysis	57
6.3.1	Mythril	57
6.3.2	Slither	57
6.4	Design Pattern Implementations	58
6.4.1	Authorization Patterns	58
6.4.2	Control Patterns	59
6.4.3	Maintenance Patterns	60
6.4.4	Security Patterns	60
7	Evaluation	61
7.1	Limitations on Selecting Patterns and Contracts	61
7.2	Security	62
7.2.1	Authorization Patterns	62
7.2.2	Control Patterns	63
7.2.3	Maintenance Patterns	64
7.2.4	Security Patterns	65
7.2.5	All Categories Together	66
7.3	Gas	67
7.3.1	Authorization Patterns	67
7.3.2	Control Patterns	67
7.3.3	Maintenance Patterns	68

7.3.4	Security Patterns	68
7.3.5	All Categories Together	69
7.4	Conclusions	69
8	Conclusion	71
8.1	Work Summary	71
8.2	Recommendations	72
8.3	Contributions	74
8.4	Limitations and Future Work	74
8.5	Personnal Overview	75
	References	77
A	Authorization Patterns	81
A.1	Access restriction	81
A.1.1	Explanation	81
A.1.2	Related patterns	81
A.1.3	Sample Code	81
A.2	Multiple authorization	82
A.2.1	Explanation	82
A.2.2	Related patterns	82
A.2.3	Sample Code	82
A.3	Ownership	84
A.3.1	Explanation	84
A.3.2	Related patterns	84
A.3.3	Sample Code	84
A.4	Roles	85
A.4.1	Explanation	85
A.4.2	Related patterns	85
A.4.3	Sample Code	85
B	Security Patterns	87
B.1	Balance Limit	87
B.1.1	Explanation	87
B.1.2	Sample code	87
B.2	Checks-effects-interaction	88
B.2.1	Explanation	88
B.2.2	Related patterns	88
B.2.3	Sample code	88
B.3	Emergency Stop	89
B.3.1	Explanation	89
B.3.2	Sample code	89
B.4	Mutex	90
B.4.1	Explanation	90
B.4.2	Related patterns	90
B.4.3	Sample code	90
B.5	Rate Limit	91
B.5.1	Explanation	91
B.5.2	Related patterns	91
B.5.3	Sample code	91

B.6	Secure ether transfer	92
B.6.1	Explanation	92
B.6.2	Related patterns	92
B.6.3	Sample code	92
B.7	Speed Bump	93
B.7.1	Explanation	93
B.7.2	Related patterns	93
B.7.3	Sample code	93
C	Control Patterns	95
C.1	Commit and reveal	95
C.1.1	Explanation	95
C.1.2	Sample code	95
C.2	Guard check	97
C.2.1	Explanation	97
C.2.2	Related patterns	97
C.2.3	Sample code	97
C.3	Memory array building	98
C.3.1	Explanation	98
C.3.2	Related patterns	98
C.3.3	Sample code	98
C.4	Oracle	99
C.4.1	Explanation	99
C.4.2	Sample code	99
C.5	Poll	100
C.5.1	Explanation	100
C.5.2	Sample code	100
C.6	Pull payment	102
C.6.1	Explanation	102
C.6.2	Related patterns	102
C.6.3	Sample code	102
C.7	Randomness	103
C.7.1	Explanation	103
C.7.2	Sample code	103
C.8	Safemath	104
C.8.1	Explanation	104
C.8.2	Sample code	104
C.9	State machine	105
C.9.1	Explanation	105
C.9.2	Sample code	105
C.10	String equality comparison	106
C.10.1	Explanation	106
C.10.2	Related patterns	106
C.10.3	Sample code	106
C.11	Tight variable packing	107
C.11.1	Explanation	107
C.11.2	Related patterns	107
C.11.3	Sample code	107
C.12	Token	108

C.12.1	Explanation	108
C.12.2	Sample code	108
C.13	Address	109
C.13.1	Explanation	109
C.13.2	Sample code	109
C.14	Byteslib	110
C.14.1	Explanation	110
C.14.2	Sample code	110
C.15	Context	111
C.15.1	Explanation	111
C.15.2	Sample code	111
D	Maintenance Patterns	113
D.1	Automatic Deprecation	113
D.1.1	Explanation	113
D.1.2	Sample code	113
D.2	Contract composer	114
D.2.1	Explanation	114
D.2.2	Sample code	114
D.3	Contract factory	115
D.3.1	Explanation	115
D.3.2	Sample code	115
D.4	Contract register	116
D.4.1	Explanation	116
D.4.2	Related patterns	116
D.4.3	Sample code	116
D.5	Contract relay	117
D.5.1	Explanation	117
D.5.2	Sample code	117
D.6	Data Segregation	118
D.6.1	Explanation	118
D.6.2	Sample code	118
D.7	Mortal	119
D.7.1	Explanation	119
D.7.2	Sample code	119
D.8	Satellite	120
D.8.1	Explanation	120
D.8.2	Sample code	120
E	R and Node.js Scripts	121
E.1	Text Mining Approach	121
E.2	Get Selected Addresses	122
E.3	Get Selected Contracts	123
E.4	Remove Duplicated Contracts	124

List of Figures

1.1	Blockchain attacks	3
2.1	Centralized vs decentralized	5
2.2	Blocks sequence	6
2.3	Blockchain transaction	6
2.4	Bitcoin transaction	8
2.5	Ethereum average gas price	12
2.6	Smart contracts	13
4.1	The innovation process	33
4.2	New concept development model	34
4.3	The AHP diagram	39
4.4	The AHP results	41
6.1	Get selected addresses algorithm	48
6.2	Get selected contracts algorithm	49
6.3	Transactions histogram	49
6.4	Text mining algorithm	50
6.5	Spreadsheet for registering of found design patterns	52

List of Tables

2.1	Ether denominations	11
2.2	Gas costs in gwei	11
2.3	Solidity variable types	15
2.4	Solidity method visibility types	15
2.5	Ethereum alternatives comparison	19
3.1	Design patterns from Wohrer and Zdun (2018b)	22
3.2	Design patterns from Wohrer and Zdun (2018a)	23
3.3	Design patterns from Xu et al. (2018)	24
3.4	Design patterns from Liu et al. (2018)	25
3.5	Design patterns from Bartoletti and Pompianu (2017)	27
3.6	Design patterns from Marchesi et al. (2020)	27
3.7	Design patterns from Volland (2019)	29
3.8	Authorization design patterns	30
3.9	Security design patterns	30
3.10	Maintenance design patterns	31
3.11	Control design patterns	31
4.1	Benefits and sacrifices of perceived value	35
4.2	Market capitalization	37
4.3	The AHP fundamental scale	39
4.4	AHP criteria comparison	40
4.5	AHP comparison for learning curve	40
4.6	AHP comparison for features	40
4.7	AHP comparison for documentation	40
4.8	AHP comparison for market share	41
5.1	Ethereum vulnerabilities	44
5.2	Design patterns categories	44
5.3	Tasks related to first objective	45
5.4	Tasks related to second objective	45
5.5	Tasks related to third objective	45
6.1	Collected contracts with more transactions	50
6.2	Text mining search results	51
6.3	Manual search results - known patterns	53
6.4	Pattern implementations by contracts transactions (thousands)	54
6.5	Manual search results - unknown patterns	55
6.6	Unknown patterns - description	55
6.7	Collected smart contracts taxonomy	55
6.8	Pattern implementations by contract type	56
6.9	Mythril results	57

6.10	Slither results	58
7.1	Authorization category - security evaluation	62
7.2	Control category - security evaluation	63
7.3	Maintenance category - security evaluation	64
7.4	Security category - security evaluation	65
7.5	All pattern categories - security evaluation	66
7.6	Authorization category - gas evaluation in wei	67
7.7	Control category - gas evaluation in wei	67
7.8	Maintenance category - gas evaluation in wei	68
7.9	Security category - gas evaluation in wei	68
7.10	All pattern categories - gas evaluation in wei	69
8.1	Patterns names found	73

List of Source Code

2.1	Solidity smart contract	14
6.1	Contract modified by access restriction pattern	59
6.2	Contract modified by guard check pattern	59
6.3	Contract modified by contract factory pattern	60
6.4	Contract modified by mutex pattern	60
A.1	Pattern: Access restriction	81
A.2	Pattern: Multiple authorization	82
A.3	Pattern: Ownership	84
A.4	Pattern: Roles	85
B.1	Pattern: Balance limit	87
B.2	Pattern: Checks-effects-interaction	88
B.3	Pattern: Emergency stop	89
B.4	Pattern: Mutex	90
B.5	Pattern: Rate limit	91
B.6	Pattern: Secure ether transfer	92
B.7	Pattern: Speed bump	93
C.1	Pattern: Commit and reveal	95
C.2	Pattern: Guard check	97
C.3	Pattern: Memory array building	98
C.4	Pattern: Oracle	99
C.5	Pattern: Poll	100
C.6	Pattern: Pull payment	102
C.7	Pattern: Randomness	103
C.8	Pattern: Safemath	104
C.9	Pattern: State machine	105
C.10	Pattern: String equality comparison	106
C.11	Pattern: Tight variable packing	107
C.12	Pattern: Token	108
C.13	Pattern: Address	109
C.14	Pattern: Byteslib	110
C.15	Pattern: Context	111
D.1	Pattern: Automatic deprecation	113
D.2	Pattern: Contract composer	114
D.3	Pattern: Contract factory	115
D.4	Pattern: Contract register	116
D.5	Pattern: Contract relay	117
D.6	Pattern: Data segregation	118
D.7	Pattern: Mortal	119
D.8	Pattern: Satellite	120
E.1	Text mining code in R	121
E.2	Get selected addresses code in Node.js	122

E.3	Get selected contracts code in Node.js	123
E.4	Remove duplicated contracts code in R	124

List of Acronyms

AHP	Analytic Hierarchy Process.
API	Application Program Interface.
BOSE	Blockchain Oriented Software Engineering.
dapps	Decentralized Applications.
DeFi	Decentralized Finance.
DoS	Denial-of-Service.
DSRM	Design Science Research Methodology.
ECDSA	Elliptic Curve Digital Signature Algorithm.
EVM	Ethereum Virtual Machine.
FFE	Fuzzy Front End.
GUI	Graphical User Interface.
ICO	Initial Coin Offering.
IDE	Integrated Development Environment.
IT	Information Technology.
LLL	Lisp Like Language.
NCD	New Concept Development Model.
NLR	Narrative Literature Review.
NPD	New Product Development.
OOP	Object-Oriented Programming.
PoS	Proof-of-Stake.
PoW	Proof-of-Work.
QR	Quick Response.
RINA	Recursive Inter-Network Architecture.
SLR	Systematic Literature Review.
SMS	Systematic Mapping Study.
TPS	Transactions Per Second.

Chapter 1

Introduction

This chapter presents the research context, followed by the description of the problem that motivated and justified the development of this work. Then, the objectives and the methodology applied to build the results are described. Finally, the structure of this dissertation is presented.

1.1 Context

Bitcoin¹ was born in 2008 from a proposal by an anonymous person using the alias of Satoshi Nakamoto. It had a massive influence on the evolution of blockchain technology, which is used as a currency to record monetary transactions between different parties in the form of sequential data blocks that are cryptographically and chronologically linked, giving birth to the concept of cryptocurrency (Nakamoto 2008).

Bitcoin's success has spawned nearly five thousand cryptocurrencies² in a new digital asset market. Transactions registered on the blockchain are immutable thanks to encryption and the peer-to-peer network, where all the constituent nodes in the network have a copy of the entire transaction history. Transactions are validated by an algorithm that also works on the network and is called a consensus mechanism. This mechanism also acts as an incentive system that pays people who buy hardware to form network nodes with cryptocurrency units. All of this works in a decentralized way, with no single points of failure, and anyone can use this technology for free (open-source). The area of finance, through cryptocurrencies, was the first in which blockchain technology was used, but in recent years projects have emerged for various areas, such as healthcare, insurance or the internet of things (D. Tapscott and A. Tapscott 2018).

The Ethereum work began in 2014 and is currently the second most famous blockchain project right behind Bitcoin. Its main objective was to put the theoretical concept of **smart contract** (Szabo 1997) into practice.

Smart contracts are programmable contracts with small clauses that when triggered create transactions that are recorded in the blockchain. Therefore, not only does Ethereum have ether, its bitcoin-like cryptocurrency, but also a smart contract development platform. Another similarity to the Bitcoin project is that Ethereum is also an open-source project that is currently supported by the largest developer community in the blockchain technology industry (Antonopoulos and Wood 2018).

¹In this document, bitcoin can appear with a lower or upper case B. The lower case means bitcoin as a cryptocurrency (abbreviation BTC has the same meaning), while upper case means Bitcoin as a blockchain and its network.

²<https://coinmarketcap.com/>

From the concept of the smart contract came the decentralized applications or dapps. These dapps, unlike traditional apps that rely on centralized web server services, have their business logic programmed into smart contracts that are deployed and stored on the blockchain in an immutable and decentralized way, as well as all transactions triggered by it (Buterin et al. 2014).

However, as blockchain and smart contract technologies are still at a very early stage of development, some bug issues and security holes have often arisen in projects implemented with these technologies. The concept of Blockchain Oriented Software Engineering (BOSE) and the first software design patterns to be applied in the development of smart contracts and dapps have been created to overcome these problems. (Porru et al. 2017; Wohrer and Zdun 2018a).

The main focus of this dissertation is precisely the analysis of design patterns for Ethereum smart contracts written in **Solidity** (detailed and described in subsection 2.2.5). Programming smart contracts with the Solidity language has been considered challenging because of its unique characteristics (Chen et al. 2017; Marchesi et al. 2020). It is currently the most popular programming language for the Ethereum platform (Antonopoulos and Wood 2018).

1.2 Problem

Ethereum smart contracts have been used to facilitate the communication between stakeholders without the need for intermediaries. Nevertheless, developing correct smart contracts has not proven to be an easy task, not only in terms of **security** (a core aspect in this work) but also in other aspects of software development like **authorization**, **maintenance** and **control** (Wohrer and Zdun 2018a). Looking for design patterns in these four categories is one of the main goals in this work. The level of **gas**³ usage also relates to this problem, as gas efficiency is also considered a crucial factor. Any modification on the Ethereum blockchain has a cost in gas which corresponds to a monetary cost. Therefore, the development of gas-efficient contracts with the help of design patterns is also a core aspect in this dissertation.

According to Zou et al. (2019): "It is hard to guarantee the security of smart contracts." Due to their financial implications, smart contracts have become a great target for cyber-attacks. The most famous example of that was the DAO attack (Luu et al. 2016), where a code vulnerability allowed the theft of 3.6 million ether (or abbreviation ETH)⁴ (approximately 50 million USD at that time). Another example of this type of hacking on Ethereum smart contracts was the Parity wallet attack where an error of a negligent chunk of programming code made stealing 500,000 ETH possible (approximately 150 million USD at that time) (Destefanis et al. 2018).

These two unfortunate episodes show that there is a lack of discipline and good practices on the recent matter called BOSE (Porru et al. 2017). Concomitantly, the demand for the development of software built on top of the Ethereum platform is rising (Wessling et al. 2018) which gives added importance to the need to support **Solidity** developers with tools and documentation to minimize the risks of their work being hacked.

³<https://ethgasstation.info/blog/what-is-gas/>

⁴<https://www.investopedia.com/tech/what-ether-it-same-ethereum/>

In order to illustrate this scenario, Figure 1.1 aims to show statistics on blockchain attacks.

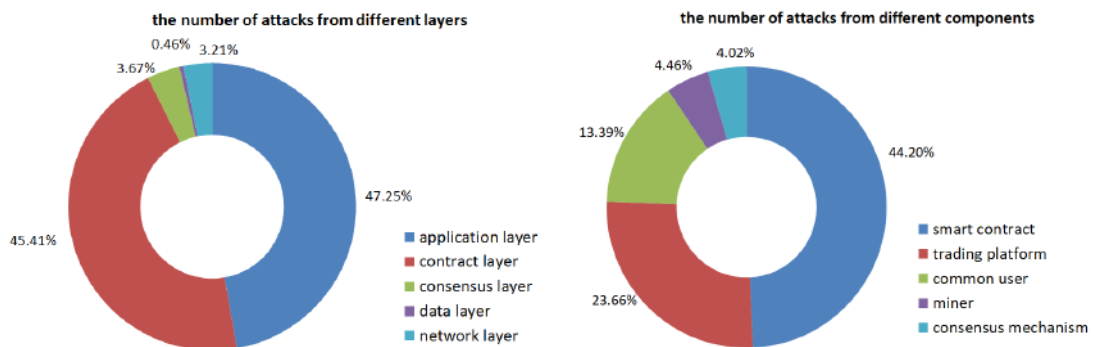


Figure 1.1: Blockchain attacks. From Huang et al. (2019).

As Figure 1.1 shows, both contract layer and contract component, are the target of a big percentage of attacks on blockchains (Ethereum and others).

This dissertation aims at helping to solve this problem in a manner that will be presented in detail in chapter 5, after the analysis of Ethereum smart contracts and the difficulties associated with their development in subsequent parts of this document.

1.3 Objectives and Research Methodology

The three objectives of this project are:

1. Collect source code of prominent Solidity contracts, following criteria defined in subsection 6.1.2, through a blockchain explorer tool (see section 2.2.6) and analyze pattern usage;
2. Analyze the existence of security vulnerabilities on selected smart contracts (see section 2.2.6), explore their pattern usage and gas consumption;
3. Examine the effect of some design patterns on modified smart contracts.

Design Science Research Methodology (DSRM) helps to implement design science research in Information Technology (IT) studies and was adopted to fulfill the mentioned objectives with the following six stages (Peffer et al. 2007):

1. **Problem identification:** characterizing the problem which is briefly presented in this first chapter in section 1.2 and described in detail in chapter 5. The value analysis of this work is done in chapter 4;
2. **Define the objectives:** presenting the objectives identified in the previous step. These objectives are presented in section 1.3 of this chapter;
3. **Design and implementation:** selecting tools and design patterns studied in the literature review (design). Looking for patterns and analyze them on the collected smart contracts (implementation). This stage is presented in chapter 6;
4. **Demonstration:** proving that the selected patterns and tools can help Solidity developers, write a set of smart contract equivalent to original contracts. This stage is presented in section 6.4 and chapter 7;

5. **Evaluation:** comparing gas usage and security vulnerabilities of the original smart contracts and those modified by some of the selected design patterns. This stage is developed in chapter 7;
6. **Communication:** sharing the problem, design patterns found and their benefits, selected development tools and the results with other researchers and audiences. This final stage will be completed, or partially completed, by presenting all the research to an evaluation committee. This document is part of the recommended communication process and offers procedures and partial solutions found in different parts of the work, including shared code in a public repository⁵.

1.4 Document Structure

This document is divided into eight chapters:

- **chapter 1** - Introduction: motivation that underlies this research, the objectives and the methodology used;
- **chapter 2** - Background: presentation of the study carried out on the main concepts used in this work;
- **chapter 3** - State of the Art: study on existing Solidity design patterns;
- **chapter 4** - Value Analysis: answering the questions of the curricular module of Value Analysis;
- **chapter 5** - Problem Statement: overview of the problem and its challenges;
- **chapter 6** - Design and Implementation: practical component of this project through the collection and analysis of Solidity smart contracts;
- **chapter 7** - Evaluation: static security and gas usage evaluation of collected smart contracts in the previous chapter;
- **chapter 8** - Conclusion: a summary of the research and suggestions for future work.

The following five appendices are also part of this document:

- **Appendix A** - Sample code of design patterns for authorization category;
- **Appendix B** - Sample code of design patterns for security category;
- **Appendix C** - Sample code of design patterns for control category;
- **Appendix D** - Sample code of design patterns for maintenance category;
- **Appendix E** - Code of R and Node.js scripts.

⁵<https://github.com/jmesmoorish/tdmei/>

Chapter 2

Background

This chapter addresses a series of relevant concepts for the understanding of this dissertation, as well as some of the core technologies addressed such as blockchain and smart contracts.

2.1 Blockchain

Blockchain is a decentralized ledger system that uses a network consensus to record, execute and validate transactions. When control is centralized on a single entity, the integrity of an information system is directly dependent on that entity, whereas in a decentralized system the integrity of information depends on all participants in the network where the system runs. Thus, to change a record, the unilateral will of one entity is not enough (Nakamoto 2008). In order to illustrate this scenario, Figure 2.1 aims to exemplify a difference.

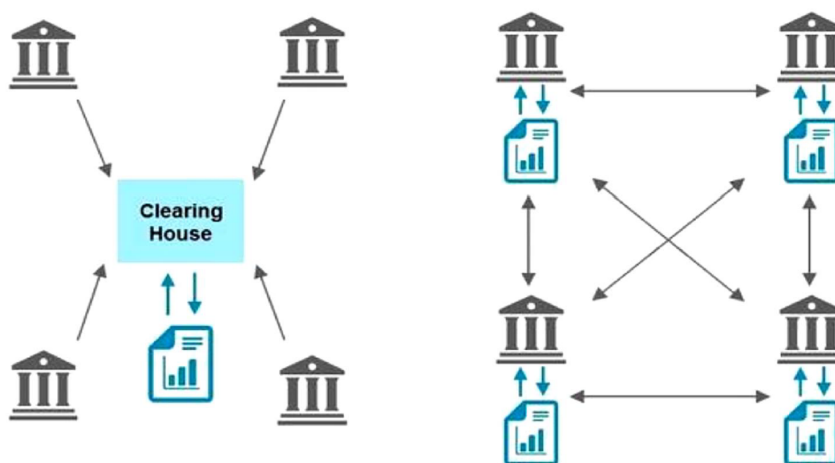


Figure 2.1: Centralized vs decentralized. From Blogs.wsj.com (2016).

On the left, information is centralized and relationships always go through a centralized point that would have the power to change information. On the right, relationships are decentralized, meaning the integrity of the information is attested by all users, in peer-to-peer¹ network. If one of them tries to tamper with an already recorded data, the others will reject the fraudulent operation (Antonopoulos 2014).

A blockchain is based on two record types: transactions and blocks. Blocks have multiple transactions that are compressed and encoded in a Merkle tree². Each block includes the

¹<https://techterms.com/definition/p2p>

²<https://blockonomi.com/merkle-tree/>

hash³ of the previous block in the blockchain, linking the two. The linked blocks form a chain (see Figure 2.2).

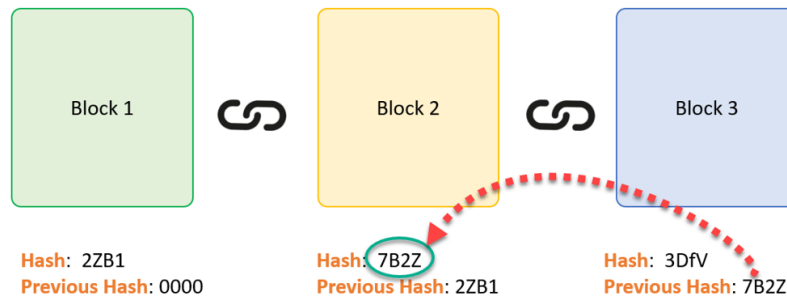


Figure 2.2: Blocks sequence. From Sqlindia.com (2019).

This iterative process confirms the integrity of the previous block and back to the first block in the chain, known as the genesis block. There is also a predefined time interval for creating a new block (in the case of the bitcoin blockchain, it lasts ten minutes) (Antonopoulos 2014).

Figure 2.3 shows the entire process of blockchain transaction.

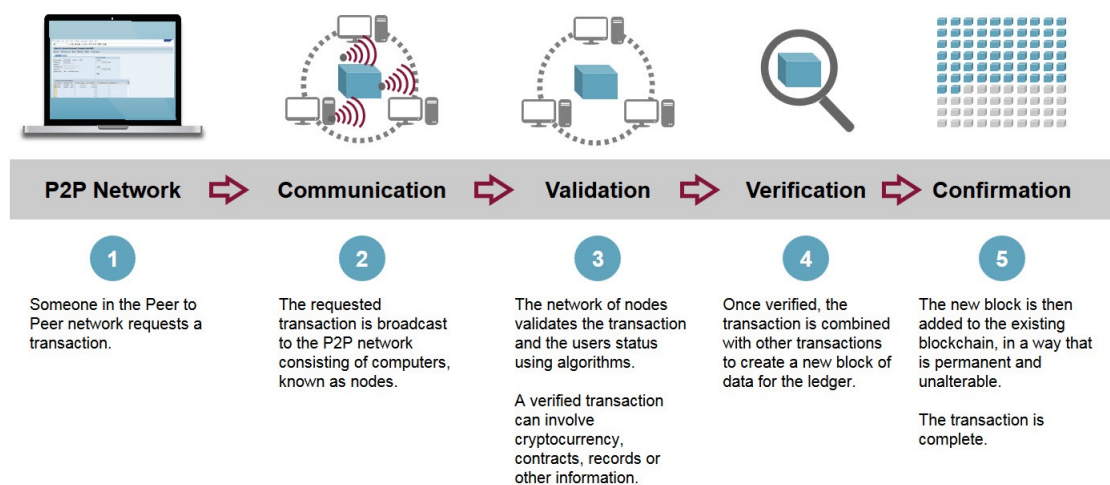


Figure 2.3: Blockchain transaction. From Msg-global.com (2019).

Blockchain-based systems have principles that distinguish them from centralized technology applications. According to A. Tapscott and D. Tapscott (2017), the five basic principles of blockchain technology are:

- **Distributed Database:** also called a distributed ledger or replicated ledger, involves having a single set of data items which are replicated and shared across multiple sites on a blockchain network;
- **Peer-to-Peer Transmission:** peer-to-peer communication is a type of computing architecture in which peers have the same privileges and authority and can communicate and interact directly with one another;

³<https://techterms.com/definition/hash>

- **Transparency with Pseudonymity:** each participant on a blockchain is represented with an address. An address is used to receive and send transactions on the blockchain. Addresses can be represented by a Quick Response (QR) code or a string of alphanumeric characters. A blockchain address is mathematically related to the public key for a wallet, as it is the hashed version of that wallet's public key;
- **Irreversibility of Records:** in the blockchain, irreversibility applies to the fact that once a block has been added to a blockchain it cannot be changed;
- **Computational Logic:** blockchain transactions only depend on computational logic because of its digital nature. The goals of blockchain-based systems are to improve efficiency and performance, as well as to eliminate a single point of failure by sharing the tasks across multiple systems.

2.1.1 Bitcoin

When talking about blockchain, it becomes almost mandatory to talk about Bitcoin, the protocol that made this technology famous.

Nakamoto's invention created not only a digital coin but also the first decentralized digital currency, i.e., an electronic money system in which there is no centralized entity entrusted with the tasks of monetary issuance and network security. In the traditional financial systems paper money is only used for small transactions and in person. For any more bulky or distant money transference, a payment system such as the banking system or PayPal⁴ is imperative. Bitcoin breaks up with this paradigm as both a digital currency and a payment system, unprecedented in world financial history (Antonopoulos 2014).

To receive, store and send bitcoins, users keep in encryption identities called addresses, which correspond to public and private keys generated through the Elliptic Curve Digital Signature Algorithm (ECDSA) (Antonopoulos 2014). Transactions are anonymous as long as addresses cannot be associated with their owners, which is usually ensured by the randomness of blockchain transfers. Unlike financial institutions, transactions are not tied to users' identity (Ammous 2018).

The Bitcoin protocol processes transactions over a distributed network using public-private cryptographic key technology. When a sender transfers funds to a recipient using the Bitcoin client in a transaction, the order is generated. The sender confirms the access for funds to be sent with his/her private key and identifies the recipient by his/her public key. In turn, the transaction request is signed with a sender private key and anyone on the network can use the sender public key to verify that the legitimate account holder has approved the request. The transaction order is included in a block with other transactions. All users running a mining bitcoin node (miners) will compete to process the transaction block. This involves solving a complicated encryption problem, which amounts to a brute force search for an essentially random string (mining) (Antonopoulos 2014).

The solution to the encryption problem ensures that the transaction block being processed is consistent with the existing block with a public record of all past transactions. Once the solution has been identified, the network can lock the block and six other nodes verify the solution (consensus mechanism). Thus, the blockchain is modified to reflect the transactions referenced in the processed block. The recipient can use their private key to generate new transactions by transferring newly received funds to someone else. Processing transactions

⁴<https://www.paypal.com/>

in this way ensures that a user does not spend the same amount twice without relying on a central authority like banks. Miners running the Bitcoin client are rewarded whenever they are the first to successfully process a transaction block and with transaction fees. This reward comes in the form of newly created bitcoin units (Antonopoulos 2014).

Figure 2.4 shows the full process of a Bitcoin transaction.

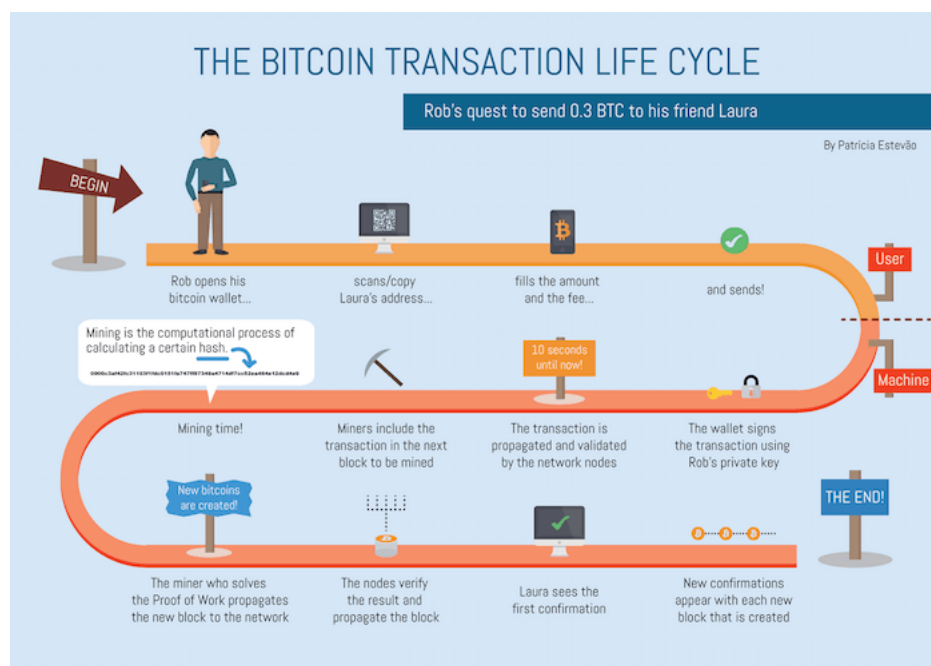


Figure 2.4: Bitcoin transaction. From Weusecoins.com (2018).

2.1.2 Consensus Mechanism

The consensus mechanism is an algorithm through which all nodes in a blockchain network reach a common agreement about the current state of the distributed ledger. In this way, consensus algorithms achieve reliability in the blockchain network and establish trust between unknown peers in a decentralized computing environment. Essentially, the consensus mechanism ensures that each new block added to the blockchain is true, valid and agreed upon by all nodes in a blockchain network. (D. Tapscott and A. Tapscott 2018)

Proof-of-Work (PoW) is the consensus algorithm used by Bitcoin and Ethereum blockchains. As it was explained in subsection 2.1.1 the main idea behind this algorithm is that nodes of the network should solve a complex mathematical puzzle and provide a solution that can be easily checked. This mathematical puzzle requires computational effort, so the nodes must find a solution to the puzzle as quickly as they can to create the new block and be rewarded for it (Ammous 2018).

There are several other consensus algorithms in other blockchain projects. One of the most famous is the Proof-of-Stake (PoS) that can replace PoW⁵ on the Ethereum platform in the future (Antonopoulos and Wood 2018).

⁵<https://tokens-economy.gitbook.io/consensus/chain-based-proof-of-stake/proof-of-stake-pos>

2.1.3 Blockchain Oriented Software Engineering

Software Engineering represents the establishment of scientific principles in conjunction with methods for the effective implementation of processes, techniques and tools for building the software product (Gamma et al. 1995).

The BOSE concept is the application of software engineering on blockchain-based systems development. The main challenges facing software engineering in the context of Blockchain oriented software are (Porru et al. 2017):

- Need for skilled software engineers to design specialized tools and techniques for blockchain guidance;
- Propose new paths from analyzing shared code repositories of blockchain-oriented applications;
- Definition of new tools for blockchain-oriented software architecture and smart contracts;
- Definition of new professional roles;
- Specialization of testing activities seeking greater rigor and safety, essentially in the context of non-functional requirements.

In this context, one of the focus of this work is to contribute to the minimization of these challenges by proposing design patterns and other tools, structured in the form of software development recommendations that can be used by Solidity developers.

2.2 Ethereum

Ethereum platform has a goal to make possible the execution of smart contracts on its blockchain. If bitcoin is uniquely considered a cryptocurrency, ether, or ETH (the native cryptocurrency of Ethereum network) is like a digital commodity since its role goes beyond a simple exchange of money between users (Buterin et al. 2014).

The most common example of Ethereum smart contracts is the Initial Coin Offering (ICO), a fundraising mechanism for project financing. The largest group of the ICOs is based on the Ethereum platform, using a token standard called ERC-20⁶. To participate in these offers, those interested should almost always acquire ether (Antonopoulos and Wood 2018).

The validation of the transactions in the network happens in a similar way to the Bitcoin network, through PoW consensus mechanism (as seen in subsection 2.1.2), which encourages, through the distribution of new currencies, that miners make investments to support the processing power of the network (Buterin et al. 2014).

The governance model of Ethereum is led by a non-profit foundation, which maintains a one hundred percent dedicated developer team on its payroll. The application development is done in a decentralized way, by hundreds of independent companies and developers around the world (Buterin et al. 2014).

Since the launch of Ethereum in 2014, several competitors have appeared in the market with a similar proposal: to create smart contracts to eliminate intermediaries in the most diverse

⁶<https://cointelegraph.com/explained/erc-20-tokens-explained/>

areas. Among the projects that have set out to accomplish this feat, stand out EOS, NEO, Stellar and Cardano (see subsection 2.2.7).

However, Ethereum is the most consolidated smart contracts and dapps platform, uninterruptedly running for over five years and concentrating a huge community from all over the world working for its development (Lielacher 2020; Mitra 2019).

Networks

Ethereum platform has a main net or **mainnet**, where real dapps are deployed making real transactions that cost real monetary value measured in gas (Antonopoulos and Wood 2018).

For development purposes, Ethereum has available test networks, where developers can test their smart contracts and dapps before the deployment on the mainnet. Transactions in test nets (**testnets**) are free and are measured in fake gas. There are three important testnets (Antonopoulos and Wood 2018):

- Rinkeby⁷;
- Ropsten⁸;
- Kovan⁹.

2.2.1 Ethereum Virtual Machine

Ethereum platform has a decentralized Turing Complete¹⁰ virtual machine, the Ethereum Virtual Machine (EVM), which runs scripts on the blockchain. Thus, it is possible to develop dapps that work exactly as programmed without any possibility of censorship, fraud or interference from third parties, because the transactions are immutable (Buterin et al. 2014).

EVM is mainly a state machine based on transactions. This state machine simply receives several inputs and, based on those inputs, will transition to a new state. A state has information stored at a specific point in time. This way the state machine acts as a computer that remembers the status of something at any given time. When there is a state change in EVM, it is similar to a blank sheet, each time something is written on the sheet, its state changes. When transactions are executed, the current state transitions to a new state (Antonopoulos and Wood 2018).

2.2.2 Accounts

The Ethereum blockchain is made up of accounts that have an address and can interact with each other. Each account has a state associated with it and an address of twenty bytes. An Ethereum address is a string of one hundred and sixty bits used to identify an account (Buterin et al. 2014).

Ethereum has two different types of accounts: external accounts and contract accounts.

External accounts: user accounts controlled by private keys.

Smart contract accounts: controlled by programming code.

⁷<https://www.rinkeby.io/>

⁸<https://github.com/ethereum/ropsten>

⁹<https://github.com/kovan-testnet/proposal>

¹⁰<https://www.binance.vision/glossary/turing-complete>

An external account can send a message to any other external account, such as a simple transfer of value and also send a message to a contract, allowing access to contract code functions. However, contract accounts cannot start new transactions on their own. Contract accounts can only trigger transactions in response to other transactions they received from an external account (Buterin et al. 2014).

2.2.3 Fees

Any calculation performed as a result of a transaction on the Ethereum blockchain has a fee charged. A gas amount is charged for all transactions that occur on the network and every time a contract is executed. Readings are not charged, but only instructions that require writing on the blockchain. A sender sets a limit and the price of gas for each transaction. Each unit of gas is measured in gwei. Since some transaction fees are small, it is more convenient to use 10 gwei instead of 0.00000001 ETH. Table 2.1 shows other units of measurement. (Antonopoulos and Wood 2018).

Table 2.1: Ether denominations. From Ethdocs.org (2016)

#	Unit	Wei value	Wei
1	wei	1 wei	1
2	Kwei (babbage)	1e3 wei	1,000
3	Mwei (lovelace)	1e6 wei	1,000,000
4	Gwei (shanon)	1e9 wei	1,000,000,000
5	microether (szabo)	1e12 wei	1,000,000,000,000
6	milliether (finney)	1e15 wei	1,000,000,000,000,000
7	ether	1e18 wei	1,000,000,000,000,000,000

Gas works as a mechanism that incentives for the miners (are rewarded with fees) and disincentives for the attackers (need to spend money to perform attacks). So, gas is a measure of computational effort. For each operation, a fixed amount of gas is assigned (for example, adding two numbers costs 3 gwei, see Table 2.2).

Table 2.2: Gas costs in gwei. From Marchesi et al. (2020)

#	Operation	Gwei	Description
1	ADD/SUB	3	Arithmetic operation.
2	JUMP	8	Unconditional jump.
3	SSTORE	5,000/20,000	Storage operation.
4	BALANCE	400	Get balance of an account.
5	CALL	25,000	Create a new account.
6	CREATE	32,000	Create a new account.

Since computing at Ethereum costs money, excessive gas consumption is usually discouraged. Each unit of gas must be paid by the sender of the transaction that triggered the computation of computational steps in a transaction to trigger the execution of a smart contract. In other words, how contracts are developed has a direct influence on their performance in terms of gas consumption. Programmers who are careless or unconcerned about gas efficiency can create excessively expensive contracts for users (Marchesi et al. 2020). On the other hand, solidity developers who try to optimize the code a lot can make reading it very difficult.

Therefore, there must be a balance between the concern with gas consumption and code readability, by programmers (Zou et al. 2019).

Gas and ether are not the same. Gas costs are fixed, while the price of ether varies according to the law of supply and demand in the cryptocurrency market. Figure 2.5 illustrates this difference.

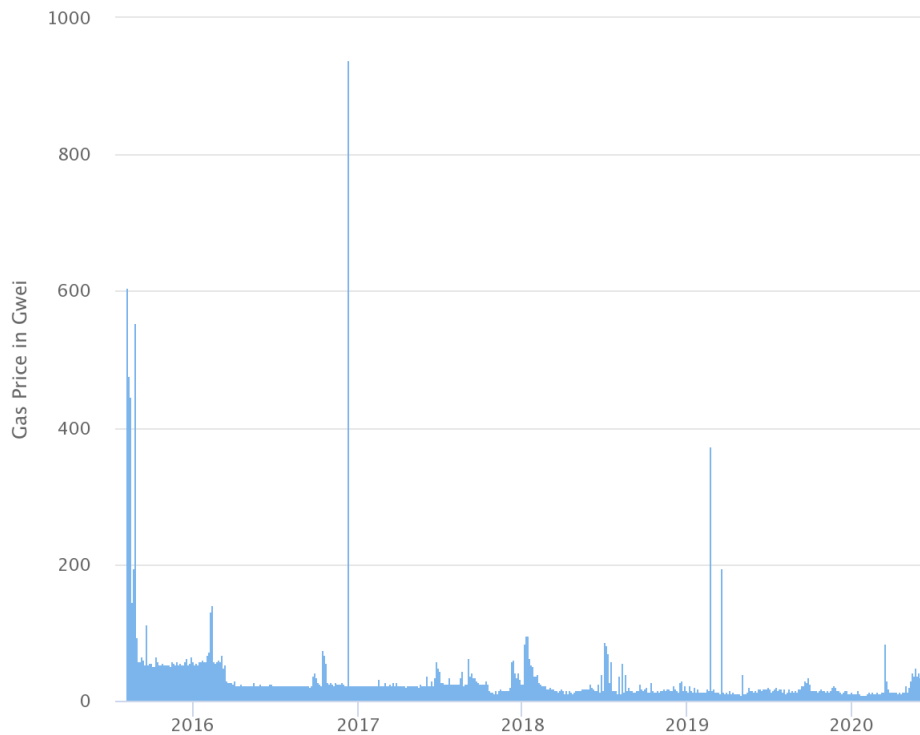


Figure 2.5: Ethereum average gas price. From Etherscan (2020)

As can be seen in Figure 2.5 gas price had some occasional peaks, but usually, it remains stable.

2.2.4 Smart Contracts

Smart contracts are computational instructions that formalize the elements of a contractual relationship and can automatically execute the terms encoded therein, once the predefined conditions are met (Cuccuru 2017).

Additionally, a smart contract can be understood as a computer protocol designed to facilitate, verify or enforce the negotiation or performance of a contract, being able to be executed or enforced on its own when a predefined condition is met. Implementation of the contract should not require direct human involvement from the moment the contract was signed (Szabo 1997).

Figure 2.6 shows how a smart contract works, illustrating an agreement between different parties.

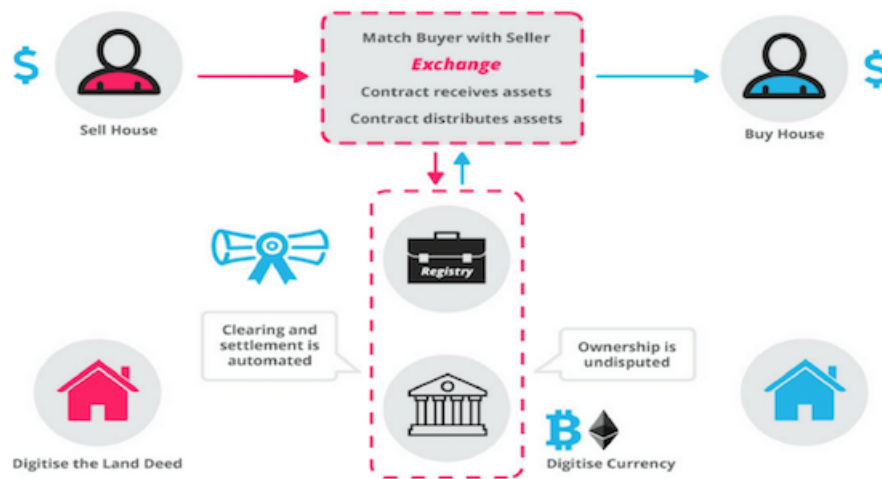


Figure 2.6: Smart contracts. From Optimum-web.com (2019).

Ethereum smart contracts roughly follow the "if x, then y" scheme. Thus, the settlement of digital relationships through the blockchain is ideal for reducing the risk of breach of any rule made in an agreement, promoting the security of value online transactions. This is because the responsibility lies with the blockchain itself to comply with the terms of the relationship, since the smart contract is launched in the blockchain, becomes independent of the will of the parties, following nothing but their self-execution instructions under the conditions encoded therein (Antonopoulos and Wood 2018).

2.2.5 Solidity

Ethereum smart contracts can be developed in several programming languages. According to Antonopoulos and Wood (2018) the following high-level languages are used to support Ethereum smart contracts:

- **Lisp Like Language (LLL):** the first high-level programming language for Ethereum smart contracts, nevertheless, it is barely used nowadays. It is a functional (declarative) programming language;
- **Serpent:** a programming language with a syntax resembling Python. It is a procedural (imperative) programming language;
- **Solidity:** nowadays, the most popular and frequently used language for Ethereum smart contracts. It is a procedural (imperative) programming language;
- **Vyper:** a programming language resembling Serpent and the most recently developed programming language for Ethereum;
- **Bamboo:** a very recent programming language influenced by Erlang¹¹. It is a procedural (imperative) programming language.

¹¹<https://www.erlang.org/>

Antonopoulos and Wood (2018) yet highlight that "... Solidity is by far the most popular, to the point of being the de facto high-level language of Ethereum and even other EVM-like blockchains."

Solidity is a high-level, contract-oriented language developed to allow the creation of Smart Contracts for the Ethereum platform. It is influenced by C++, Python and Javascript, and the code written with this language has to be compiled to run on the EVM (Ethereum.org 2019).

When smart contracts are developed, the first paradigm shift begins with this statement: "The code is law". For programmers who are used to developing traditional centralized applications and making code changes with some frequency, it is new to know that, once the code of a smart contract is deployed on the blockchain, it can never be changed. So, Solidity smart contracts development is composed of three main steps (Ethereum.org 2019):

1. **Implementation:** the writing of the smart contract code itself;
2. **Testing:** the writing of unit tests code to ensure that the smart contract will work without issues. This step is not mandatory but is highly recommendable due to the immutable nature of blockchain technology;
3. **Deployment:** the launch of the smart contract code in the form of an immutable transaction on the blockchain.

The structure of a smart contract on Solidity (which file extension is *.sol) is similar to a class on an Object-Oriented Programming (OOP) language. Solidity is strongly typed and has inheritance as a feature. (see Listing 2.1).

```
1 //this is a comment
2 pragma solidity >= 0.4.0 < 0.6.0; // version of the solidity compiler
3
4 //***** Contract Begin *****/
5
6 contract SimpleStorage {
7     uint storedData;
8     function set(uint x) {
9         storedData = x;
10    }
11    function get() constant returns (uint retVal) {
12        return storedData;
13    }
14 }
15
16 //***** Contract End *****/
```

Listing 2.1: Solidity smart contract. Adapted from Ethereum.org (2019).

Variables

Most variable types on Solidity are known from elder programming languages but there is also a new type, the address type, which is specific to and needed for smart contracts programming. Table 2.3 shows some basic types of variables on Solidity (Ethereum.org 2019).

Table 2.3: Solidity variable types.

#	Type	Description
1	string	Sequence of characters.
2	bool	Boolean value.
3	int	Integer, positive or negative. Has no decimal.
4	uint	Unsigned integer, positive number. Has no decimal.
5	fixed	Fixed point number, positive or negative. Has decimal.
6	ufixed	Unsigned fixed point number, positive number. Has decimal.
7	address	Has methods tied to it for sending monetary funds (ether).

Method visibility

Solidity also has several different types of method visibility, most are similar to method visibility types of other programming languages. The pure and payable types are specific to Solidity. Table 2.4 shows some basic types of method visibility on Solidity (Ethereum.org 2019).

Table 2.4: Solidity method visibility types.

#	Type	Description
1	public	Anyone can call this function.
2	private	Only this contract can call this function.
3	view	This function returns data and does not modify the contract's data.
4	pure	Function will not modify or even read the contract's data.
5	payable	When someone calls this function he/she might send ether along.

2.2.6 Tools

The development of any project requires the use of tools. In this work, they are divided into two groups: references and software tools. The former are used for background support and to understand what has been done on the subject so far to avoid duplication. The latter involves some instruments without which it would be difficult or even impossible to go forward with the project. That is the case of two types of software tools that are used on this work, namely tools like a blockchain explorer to collect contracts and tools to check the security of smart contracts code.

Blockchain Explorer

The fact that Ethereum blockchain is public¹² means anyone can see all transactions at any time. This is possible due blockchain explorers, which allow the consultation, not only of all transaction history but also of the code of any smart contract that was deployed on the blockchain. Therefore, this kind of tool is very useful to the development of this dissertation because it allows the fulfillment of the first goal of this work (see section 1.3).

¹²<https://www.binance.vision/blockchain/private-public-and-consortium-blockchains-whats-the-difference>

Three options of blockchain explorers were analyzed and compared, namely:

- Etherscan¹³;
- Etherchain¹⁴;
- BlockScout¹⁵.

Etherscan

Etherscan is the most popular and oldest of the three options. The most important features are:

- It allows searches by address, transaction hash, block, tags, labels, websites and more;
- It offers all kinds of information about ether and other crypto-assets (balances, prices, trading volume and more);
- It has services for developers including an Application Program Interface (API) that allows the download of smart contracts code (the most important feature for this work).

Etherchain

Even though Etherchain is very similar to Etherscan, it presents some important differences:

- It does not have an API for developers;
- It has a more pleasant and colourful Graphical User Interface (GUI) than Etherscan;
- It offers information about mining pools¹⁶, Ethereum economy and network usage.

BlockScout

BlockScout is the most recent Ethereum blockchain explorer and is slightly different from Etherscan and Etherchain:

- It allows the switching between Ethereum and other blockchains;
- It offers block-oriented information as a result of queries of the users;
- It is open-source¹⁷.

Comparison

All three Ethereum blockchain explorers work in a similar way. In short:

1. **Etherscan**: it is the oldest and most famous (more web info);
2. **BlockScout**: it is open-source, allows other blockchains monitoring;
3. **Etherchain**: it has the most pleasant GUI.

¹³<https://etherscan.io/>

¹⁴<https://www.etherchain.org/>

¹⁵<https://blockscout.com/>

¹⁶<https://www.investopedia.com/terms/m/mining-pool.asp>

¹⁷<https://github.com/poanetwork/blockscout/>

Security

Security tools are used to statically and automatically analyze smart contracts code. Durieux et al. (2019) have selected nine automated security tools and analyzed 47,587 Ethereum smart contracts. The two tools with the best results, either individually or in combination, were as follows:

- **Mythril**¹⁸: it is based on taint analysis, concolic analysis and control flow checking of the EVM bytecode, to search for data that make possible the discovery of vulnerabilities in the Solidity smart contracts;
- **Slither**¹⁹: it is a static analysis framework that converts Solidity smart contracts into an intermediate representation called SlithIR to runs a suite of vulnerability detectors and print visual information about security failures of the smart contracts.

2.2.7 Alternative Platforms

As it was already explained at the end of section 2.2, there are several alternatives to the Ethereum platform. This section presents an overview of some of them, namely EOS²⁰, Neo²¹, Stellar²² and Cardano²³. All of these platforms were chosen as an alternative to Ethereum because of the following features (CoinMarketCap.com 2020):

- Top thirty in terms of market value;
- Cryptocurrency associated. Ethereum has ETH, EOS blockchain has EOS asset, NEO blockchain has NEO asset and Cardano has ADA;
- Open-source platforms.

EOS

The goal of EOS is to create a network that will serve as the basis for other companies to install their information systems and use the blockchain without having to build a decentralized structure from the start. According to their development team, companies have not yet begun to use decentralized technologies because it is too complex and requires an expenditure of energy, time and money to create something that is not their main activity. Besides, decentralized network operations are very slow compared to centralized services. For example, the Bitcoin network carries out around seven transactions per second, Ethereum fifteen, but companies need speeds in the tens of thousands. If blockchain does not allow this, it is impracticable to transfer their structures to decentralized networks (Grigg 2017).

So EOS proposes two things (Grigg 2017):

- A platform in which companies can plug their systems and enjoy the benefits of a decentralized network as they need it;
- The delivery of a network that is capable of performing thousands of exchanges per second, such as Visa²⁴, which runs 20,000 transactions in that fraction of time.

¹⁸<https://github.com/ConsenSys/mythril/>

¹⁹<https://github.com/crytic/slither/>

²⁰<https://eos.io/>

²¹<https://neo.org/>

²²<https://www.stellar.org/>

²³<https://www.cardano.org/>

²⁴<https://usa.visa.com/>

NEO

Much more than just copying Ethereum and implementing it primarily in China, NEO intends to deliver a somewhat broader view than the one of Vitalik Buterin, creator of Ethereum. NEO aims to foster a new economic model through which financial resources could be managed more safely and efficiently (Rosic 2018a). According to their development team, this model would be a "smart economy" capable of preventing fraudulent cases and corruption. Their definition of "smart economy" is presented as the digitization of the whole economy, including money, physical and intangible assets, identification of people, belongings and machines. With this in mind, NEO is designed to be an open network of dapps that support the "smart economy" proposed by its development team (Rosic 2018a).

Stellar

Stellar has a strategy of key partnerships and incentives for new users with a focus on users of developing countries. The project was born in 2014 and its main objective is to be a payment platform for companies that need to transmit resources between countries in a matter of seconds (Stellar.org 2018).

The heaviest partnership of Stellar, IBM²⁵, has already stated, through its head of the blockchain division, Jesse Lund, that this partnership has a strategic purpose with a future business that this company develops. According to Lund, the Stellar platform is the backbone of IBM's universal payment solution and for him this is the company's bet to carry out all types of settlement and payment of any company. This would include much more assets than just cryptocurrencies since the Stellar platform allows for the creation of other crypto assets (Stellar.org 2018).

Cardano

Cardano platform calls itself the third generation of the blockchain - for having studied in depth the first generation (blockchain 1.0), initiated by Bitcoin, and the second, proposed by Ethereum (blockchain 2.0). In quite a simple way, the first phase of this technology allowed people to use the network to transact monetary value, using Bitcoin (Rosic 2018b).

The second allowed the scope of this decentralization technology to expand beyond money. It was Ethereum who proposed a new application for the blockchain, functioning as a huge decentralized computer. It was with these two views that the Cardano team proposed the third generation (blockchain 3.0), which would solve the existing problems in the previous networks (Rosic 2018b).

Bitcoin and Ethereum have scalability issues reported by those who try to make a big number of transactions on these platforms. This problem in the case of Bitcoin made a lot of sense in late 2017 when the value of this cryptocurrency reached its maximum value²⁶. In the case of Ethereum this problem was noticed when the dapp cryptokitties²⁷ started to operate inside the network and made the transactions slower (Rosic 2018b).

Cardano proposes to solve this problem of scalability by using a technology known as Recursive Inter-Network Architecture (RINA). RINA allows the existence of several networks

²⁵<https://www.ibm.com/>

²⁶<https://www.businessinsider.com/bitcoin-payment-mining-fees-hit-new-high-2017-12>

²⁷<https://www.cryptokitties.co/>

encompassing several people communicating with each other and if necessary, the exchange of information between networks (Rosic 2018b).

Ethereum Alternatives Comparison

Table 2.5 shows a comparison between Ethereum and its alternatives in terms of relevant features on blockchain platforms, namely: issue date (when the code has made available on GitHub), consensus method, Transactions Per Second (TPS) and block speed (in seconds). It should be noted that all these blockchains are public or permissionless, which means that anyone can join the blockchain network. However, private or permissioned blockchains like Hyperledger²⁸ restrict participation in their network.

Table 2.5: Ethereum alternatives comparison. Adapted from BinanceInfo (2020)

#	Platform	Issue Date	Consensus	TPS	Block Speed
1	Ethereum	2014-07-24	PoW	15	15s
2	EOS	2017-07-02	Delegated PoS	6000	1s
3	Neo	2014-06-01	Delegated Byzantine Fault Tolerance	30	15s
4	Cardano	2017-10-02	PoS	250	20s
5	Stellar	2014-08-01	Stellar Consensus Protocol	1000	6s

As one can observe in Table 2.5, EOS stands above from the others with a superior TPS. Nevertheless, at present, Ethereum has yet the *first-mover* advantage, one of the reasons why Ethereum is still the most popular smart contracts platform (Finance-monthly 2019).

²⁸<https://www.hyperledger.org/>

Chapter 3

State of the Art

This chapter presents one of the main tasks of this work - an exhaustive study of design patterns on Solidity smart contracts. These patterns are to be applied to the creation of reusable smart contracts to avoid bad practices, gas consumption issues and security vulnerabilities.

According to Gamma et al. (1995), "A design pattern names, abstracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.". Generally, most design patterns are abstract and language-agnostic, but in the case of Ethereum as it is yet a very recent platform, most literature and web material are related to Solidity programming language (the most popular and used in Ethereum). So, this chapter is mostly about Solidity design patterns.

3.1 Literature Research

Literature review can be carried out using well-known and studied methodologies. For this work, the following three types of review were analyzed:

- **Narrative Literature Review (NLR):** it is often used for the theoretical foundation of articles, dissertations, theses and course completion papers, but it does not apply sophisticated and exhaustive search strategies.(Torgerson 2003);
- **Systematic Literature Review (SLR):** it is used for scientific research through systematic and explicit methods to retrieve, select and evaluate the results of relevant studies (Torgerson 2003);
- **Systematic Mapping Study (SMS):** it is used for studies that try to find what is the current status in a given topic in general terms. These studies generally try to verify who are the authors who publish the most and the institutions, years of publication and research methods involved (Sampaio 2015).

SMS was the type of literature review selected for this work because it was considered the most appropriate to the nature of this dissertation and its main theme - Ethereum, a still very young smart contracts platform, with an even more recent set of specific publication. It is important to filter the year of publication among other aspects. Therefore, the selection of papers to carry out this study was based on the three following criteria:

- **Published after 2016:** as the Ethereum platform is relatively young (started in 2014), it makes sense that this study is also based on late papers;
- **With five citations or more:** a way to only choose papers that already have some confirmed credit;
- **Centered on Solidity and Ethereum:** as software design patterns is a very comprehensive theme, the words Ethereum and Solidity were chosen to filter content.

In the project from Wohrer and Zdun (2018b), a data analysis was performed using the **Grounded Theory** (Charmaz and Belgrave 2007) technique in order to discover security design patterns. This research was based on the search for design patterns that are most prevalent in smart contracts programmed in Solidity and what security problem these design patterns intend to solve. The authors point out that the Ethereum platform and the Solidity language evolve very quickly at the level of newly added statements, bugs and discovered security risks. Therefore, smart contract developers should be very aware of any changes and aware that they may be required to refactor their smart contract code at any time.

The main aspect to be analyzed in this paper was security, because today, most developed smart contracts belong to dapps that operate in the finance area, performing many transactions of considerable monetary value. Therefore, it is very important that smart contract development is methodical to avoid financial losses due to code bugs. The data sources analyzed in this paper are multiple, ranging from Solidity official documentation to blogs and internet discussion forums and ending with code reading of existing smart contracts. Table 3.1 shows the six design patterns analyzed in this paper.

Table 3.1: Design patterns from Wohrer and Zdun (2018b).

#	Pattern	Category	Problem	Solution
1	Balance limit	Security	In the case of security bug discovery by hackers, all contract funds can be lost.	Set a contract balance limit at risk.
2	Checks-effects-interaction	Security	Reentrancy vulnerability. External calls make a contract not work correctly.	Follow a recommended functional code order.
3	Emergency stop	Security	In the case of security bug discovery, a contract must be stopped.	Define contract stoppage functionality.
4	Mutex	Security	Reentrancy vulnerability. External calls make a contract not work correctly.	Set a locking boolean variable.
5	Rate limit	Security	A smart contract can break when there is a simultaneous execution of many heavy tasks.	Regulate how many times a task can be executed in a time interval.
6	Speed bump	Security	A smart contract can break when there is a simultaneous execution of many heavy tasks.	Prolong completion of multiple heavy tasks.

The authors of the previous paper extended their work with another article (Wohrer and Zdun 2018a), this time to find design patterns for more categories beyond security. Here

the survey methodology was again (as in previous work) the **Grounded Theory** technique, but now combined with **Multivocal Literature Research** (Garousi, Felderer, and Mäntylä 2016), which "... includes grey literature (e.g., blogs, videos and web pages) in addition to published white literature (e.g., academic journals and conference papers)". This paper presents twelve design patterns of four categories, namely *action and control*, *authorization*, *lifecycle* and *maintenance*. Table 3.2 shows them.

Table 3.2: Design patterns from Wohrer and Zdun (2018a).

#	Pattern	Category	Problem	Solution
1	Commit and reveal	Action and Control	Some contracts need data confidentiality.	Give confidentiality to data transactions, during a certain period of time after which data are revealed.
2	Oracle	Action and Control	Some contracts need off-chain data.	Access external information by requesting it to a data carrier located off-chain.
3	Pull payment	Action and Control	Fund transfers can fail.	Let payment receivers withdraw the funds.
4	State machine	Action and Control	Some dapps have multi-scenario logic.	Adapt contracts to several different stages and transitions.
5	Access restriction	Authorization	Any party can call any contract function.	Ensure specific tasks are executed only if some predefined requirements are met.
6	Ownership	Authorization	Any party can call any contract function.	Restrict specific sensitive tasks only to the contract owner.
7	Automatic deprecation	Lifecycle	Some contract methods are no longer needed, after a period.	Set a time interval to deprecate a contract method.
8	Mortal	Lifecycle	Some contracts are no longer needed, after a period.	Give to a contract self-destruction functionality.
9	Contract register	Maintenance	Contract participants need to know its latest version.	Make possible to participants querying the address of the latest version of a contract.
10	Contract relay	Maintenance	Contract participants need to know its latest version.	Create a contract that acts as a proxy that relays all requests to the latest version of a contract.
11	Data segregation	Maintenance	In updates of a contract, data need to migrate to the new version.	Separate contract data in another contract, avoiding data migrations when a new version is created.
12	Satellite	Maintenance	Deployed contracts are immutable, which hinders contract upgrades.	Get around blockchain immutability, by adding new functionalities to a contract through auxiliary contracts deployment.

The eighteen aforementioned patterns are created automatically in a project by a code generator and a domain-specific language (Wohrer and Zdun 2020).

As for the work of Xu et al. (2018), it presents a set of design patterns (general concept, not only Ethereum smart contracts) that focuses on the integration of blockchain technology in traditional applications. One of the researched design patterns categories is called *interaction with external world*, which includes three design patterns. For the authors, a blockchain-based system also needs to manage data that can be handled on-chain or off-chain, depending on the needs and business logic of the system. Thus, four design patterns from the *data management* category were scrutinized. Another concern of the authors was the *security* category mainly concerning the management of private keys belonging to the blockchain-based system participants. In this category three design patterns were analysed. The last category analyzed in this paper is *structural*, where five design patterns were assigned. Here the focus of the study was on the dependencies and behaviors between smart contracts and the issue of updating contract versions. Table 3.3 shows the fifteen design patterns analyzed in this paper.

Table 3.3: Design patterns from Xu et al. (2018).

Begin of Table 3.3				
#	Pattern	Category	Problem	Solution
1	Encrypting on-chain data	Data management	Equals to Commit and reveal pattern described on previous study.	
2	Off-chain data storage	Data management	On-chain data storage is gas-costly.	Store data off-chain and ensure its integrity through hashing saved on-chain.
3	State channel	Data management	Some transactions have less monetary value than their fees.	Create off-chain channels for micro-transactions that have less monetary value than fees.
4	Tokenisation	Data management	Management of tokens is one of the most often tasks in Ethereum.	Use a standard contract to manage tokens.
5	Legal and smart contract pair	Interaction with external world	Many smart contracts correspond to real contracts.	Bind contracts to corresponding legal agreements.
6	Oracle	Interaction with external world	Same design pattern described on previous study.	
7	Reverse oracle	Interaction with external world	Sometimes, off-chain systems need to get on-chain data.	Make smart contracts acting like an oracle that responds to off-chain systems requests.
8	Multiple authorization	Security	Any party can call any contract function.	Restrict specific tasks only to a group of participants.
9	Off-chain secret enabled dynamic authorization	Security	Any party can call any contract function.	Bind transactions authorization to an off-chain hash.

Continuation of Table 3.3				
#	Pattern	Category	Problem	Solution
10	X-confirmation	Security	Transactions need multiple confirmations to be added to the blockchain.	Ensure that a transaction added into blockchain is immutable with high probability, by counting x block confirmations.
11	Contract registry	Structural	Equals to Contract register pattern described on previous study.	
12	Data contract	Structural	Equals to Data segregation pattern described on previous study.	
13	Embedded permission	Structural	Equals to Access restriction pattern described on previous study.	
14	Factory contract	Structural	Sometimes is needed to create hundreds or thousands of instances of a contract.	Create a contract to generate new instances of another contract automatically.
15	Incentive execution	Structural	Occasionally contract functions are almost never invoked.	Reward a participant for calling a certain task of the contract.
End of Table 3.3				

Patterns **Reverse oracle**, **Legal and smart contract pair**, **Off-chain data storage**, **State channel**, **Off-chain secret enabled dynamic authorization**, **X-confirmation** and **Incentive execution** were not selected for this project (in the practical component) because they were projected for blockchain general purposes and are not fitted to Solidity smart contracts.

Liu et al. (2018), present eight smart contract design patterns for Solidity, divided into four categories: *creational*, *structural*, *inter-behavioral* and *intra-behavioral*. An approach that seeks to find design patterns for smart contracts was used, from known design patterns commonly used in distributed and peer-to-peer systems, as well as in software design patterns in general. The eight design patterns researched in this work were applied in a case study of a blockchain-based system for product tracking during production and distribution. Table 3.4 shows the eight design patterns analyzed in this project.

Table 3.4: Design patterns from Liu et al. (2018).

Begin of Table 3.4				
#	Pattern	Category	Problem	Solution
1	Contract composer	Creational	Big contracts become hard to read and understand.	Create auxiliary contracts to act like components of complex contracts.
2	Contract factory	Creational	Equals to Factory contract pattern described on previous study.	

Continuation of Table 3.4				
#	Pattern	Category	Problem	Solution
3	Contract mediator	Inter-behavioral	Some times two contracts or more have to communicate often.	Ease communication between different contracts, by encapsulation of interactions from one contract to others.
4	Contract observer	Inter-behavioral	Equals to Contract relay pattern described on previous study.	
5	Hash secret	Intra-behavioral	Equals to Commit and reveal pattern described on previous study.	
6	Multi-signature	Intra-behavioral	Equals to Multiple authorization pattern described on previous study.	
7	Contract decorator	Structural	Equals to Satellite pattern described on previous study.	
8	Contract facade	Structural	Sometimes several contracts have similar functions.	Provide simple interfaces from contract addresses, in order to similar contracts share functions.
End of Table 3.4				

Patterns **Contract facade** and **Contract mediator** were not selected for this project (in the practical component) because they were not found in web-pages related to Solidity development.

In the paper from Bartoletti and Pompianu (2017), a qualitative analysis of smart contracts was made not only for Ethereum but also for Bitcoin blockchain. The purpose of this analysis was to catalog the smart contracts analyzed according to application and design patterns. Regarding the design pattern aspect (for Ethereum smart contracts), which is relevant to this for this dissertation, nine patterns were found.

Unlike the works mentioned above, these nine design patterns were not divided into categories and most of them had also been referenced by the works described above. The methodology used can be described in the following terms: "... for Ethereum, we collect on January 1st, 2017 all the contracts marked as "verified" on the blockchain explorer etherscan.io. This means that the contract bytecode stored on the blockchain matches the source code (generally written in a high-level language, such as Solidity) submitted to the explorer. In this way, we obtain a sample of 811 contracts.". Table 3.5 shows the nine design patterns found in this work.

Table 3.5: Design patterns from Bartoletti and Pompianu (2017).

#	Pattern	Problem	Solution
1	Authorization	Equals to Ownership pattern described on previous study.	
2	Fork check	Ethereum blockchain was forked once.	Check if a contract is running on the main chain and not on a forked chain.
3	Math	Math operations require validations in contracts.	Encode logic that handles the execution of some regular math instructions.
4	Oracle	Same design pattern described in the previous study.	
5	Poll	Voting systems are common in contracts.	Allow users to vote on some matter in a reusable way.
6	Randomness	Generation of random values on-chain is not an easy and quick task.	Query and get random values from an off-chain service.
7	Termination	Equals to Mortal pattern described on previous study.	
8	Time constraint	Equals to Access restriction pattern described on previous study.	
9	Token	Equals to Tokenisation pattern described on previous study.	

Pattern **Fork check** was not selected for this project (in the practical component) because it was not found in web-pages related to Solidity development.

In the work of Marchesi et al. (2020), a study was made on exclusively design patterns with a focus on gas optimization. With a research based on blogs and discussion forums on Ethereum and analysis of existing Solidity smart contracts, twenty-four patterns were analyzed, divided into five categories, namely: *external transactions*, *storage*, *saving space*, *operations* and *miscellaneous*. Table 3.6 summarizes it.

Table 3.6: Design patterns from Marchesi et al. (2020).

Begin of Table 3.6				
#	Pattern	Category	Problem	Solution
1	Data contract	External transactions	Equals to Data segregation pattern described on previous study.	
2	Event log	External transactions	Store event logs on the blockchain is expensive.	Let off-chain systems straightly access event logs on the blockchain.
3	Proxy	External transactions	Equals to Contract relay pattern described on previous study.	

Continuation of Table 3.6				
#	Pattern	Category	Problem	Solution
4	Freeing storage	Miscellaneous	Some storage variables are no longer needed after a time.	Use keyword <i>delete</i> on no longer needed data and get gas refunds for it.
5	Optimizer	Miscellaneous	Optimize Solidity code to save gas is not easy.	Activate the Solidity optimizer on compilers.
6	Avoid redundant operations	Operations	Operations cost gas.	Shun redundant operations.
7	Fewer functions	Operations	Methods implementation costs gas.	Do not create methods in an excessive way.
8	Internal function calls	Operations	<i>Public</i> methods are more expensive than <i>internal</i> methods.	Use <i>internal</i> methods instead of <i>public</i> ones as much as possible.
9	Limit external calls	Operations	Calls to external contracts are expensive.	Avoid external calls as much as possible.
10	Limit modifiers	Operations	Modifiers increase the size of methods, costing more gas.	Use modifiers in a balanced way.
11	Short circuit	Operations	Operations cost gas.	Use logical operators (&& ,), in a way to minimize the probability of execution the second expression.
12	Short constant strings	Operations	Storing strings costs gas.	Use small-sized constant strings.
13	Single line swap	Operations	Setting value in a variable costs gas.	Use (a, b) = (b, a) instead of using an auxiliary variable.
14	Use libraries	Operations	The bigger a contract , the more expensive it is.	Use libraries in a balanced way to save gas.
15	Write values	Operations	Operations cost gas.	Write values instead of calculating them, every time it is possible.
16	Default value	Saving Space	Initialize variables costs gas.	Do not initialize variables if its initial value is zero. By default, all variables are initialized to zero.
17	Explicitly mark external function	Saving Space	Input arguments of <i>public</i> methods are copied to memory, and it costs gas.	Input arguments of <i>external</i> methods are read directly from <i>calldata</i> memory. So, set as <i>external</i> , methods called only externally.
18	Fixed size	Saving Space	Variable-size data are more expensive than fixed-size.	Use fixed-size variables as much as possible.

Continuation of Table 3.6				
#	Pattern	Category	Problem	Solution
19	Mapping vs array	Saving Space	Mappings are less gas-costing, whilst arrays are iterable and packable.	Use mappings to create lists, unless it is necessary to iterate data or it is advantageous to struct and pack variables.
20	Minimize on-chain data	Saving Space	Storage (persistent data) is much more expensive than memory (volatile data).	Use memory instead of storage as much as possible.
21	Uint* vs Uint256	Saving Space	At a time, EVM runs uint* (unsigned integers smaller than 256 bits), which are converted to uint256 and it costs extra gas.	Use uints of 128 bits or less, when packing several variables in one slot. Otherwise, use uint256.
22	Limit storage	Storage	Storage (persistent memory) is the most expensive type of memory on Ethereum.	Limit this type of memory only to the essential and limit the changes on it.
23	Packing booleans	Storage	Booleans are stored as unsigned integers of 8 bits. Nevertheless, 1 bit would be sufficient.	Pack booleans in a unique unsigned integer of 256 bits.
24	Packing variables	Storage	The minimum unit of memory is a key-value pair with keys and values of 32 bytes each.	Pack statically-sized variables consecutively in a gas optimized way.
End of Table 3.6				

Patterns **limit storage** and **packing variables** were the selected patterns (from Table 3.6) to the practical component of this dissertation because their code was the most easily found on web-pages related to Solidity development.

Many of the aforementioned patterns are easily confirmed on web-pages related to Solidity development. Throughout this part of the research three other design patterns arose, even though they were not in the reviewed papers (Volland 2019) (see Table 3.7).

Table 3.7: Design patterns from Volland (2019).

#	Pattern	Category	Problem	Solution
1	Guard check	Control	User inputs can have unexpected values.	Validate data of user inputs.
2	String equality comparison	Control	Strings comparisons are expensive.	Check the equality of two strings in a gas economic way.
3	Secure ether transfer	Security	Ether transfers are a sensitive task.	Transfer ether between addresses in a secure way.

As a result of this study seventy-seven design patterns were found, fifteen repetitions were considered and thirty-two patterns were not selected, thus establishing a total of thirty selected design patterns. The division by category adopted in this research is similar to the first two studies from Wohrer and Zdun, namely **Authorization** (three patterns), **Control** (twelve patterns), **Maintenance** (eight patterns) and **Security** (seven patterns).

3.2 Authorization Patterns

Three patterns were selected in this category. Table 3.8 shows them.

Table 3.8: Authorization design patterns.

#	Name	Other Names	Reference	Sample
1	Access restriction	Restriction access; Embedded permission; Time constraint.	Wohrer and Zdun (2018a)	section A.1
2	Multiple authorization	Multi-signature.	Xu et al. (2018)	section A.2
3	Ownership	Authorization.	Wohrer and Zdun (2018a)	section A.3

3.3 Security Patterns

Seven patterns were selected in this category. Table 3.9 shows them.

Table 3.9: Security design patterns.

#	Name	Other Names	Reference	Sample
1	Balance limit		Wohrer and Zdun (2018b)	section B.1
2	Checks-effects-interaction		Wohrer and Zdun (2018b)	section B.2
3	Emergency stop	Circuit breaker; Paus- able.	Wohrer and Zdun (2018b)	section B.3
4	Mutex	Reentrancy guard.	Wohrer and Zdun (2018b)	section B.4
5	Rate limit		Wohrer and Zdun (2018b)	section B.5
6	Secure ether transfer		Volland (2019)	section B.6
7	Speed bump		Wohrer and Zdun (2018b)	section B.7

Most security patterns aim to eliminate vulnerabilities in smart contracts. Therefore, it is expected that its implementation has this effect on the experiences carried out in this work (described and detailed in subsection 7.2.4 and subsection 7.2.5).

3.4 Maintenance Patterns

Eight patterns were selected in this category. Table 3.10 shows them.

Table 3.10: Maintenance design patterns.

#	Name	Other Names	Reference	Sample
1	Automatic deprecation		Wohrer and Zdun (2018a)	section D.1
2	Contract composer		Liu et al. (2018)	section D.2
3	Contract factory	Factory contract.	Liu et al. (2018)	section D.3
4	Contract register	Contract registry.	Wohrer and Zdun (2018a)	section D.4
5	Contract relay	Proxy; Proxy delegate; Contract observer.	Wohrer and Zdun (2018a)	section D.5
6	Data segregation	Data contract; Eternal storage.	Wohrer and Zdun (2018a)	section D.6
7	Mortal	Termination.	Wohrer and Zdun (2018a)	section D.7
8	Satellite	Contract decorator.	Wohrer and Zdun (2018a)	section D.8

3.5 Control Patterns

Twelve patterns were selected in this category. Table 3.11 shows them.

Table 3.11: Control design patterns.

#	Name	Other Names	Reference	Sample
1	Commit and reveal	Encrypting on-chain data; Hash secret.	Wohrer and Zdun (2018a)	section C.1
2	Guard check		Volland (2019)	section C.2
3	Memory array building	Limit storage.	Marchesi et al. (2020)	section C.3
4	Oracle		Wohrer and Zdun (2018a)	section C.4
5	Poll		Bartoletti and Pompianu (2017)	section C.5
6	Pull payment	Pull over push; Withdrawal contract.	Wohrer and Zdun (2018a)	section C.6
7	Randomness		Bartoletti and Pompianu (2017)	section C.7
8	Safemath	Math.	Bartoletti and Pompianu (2017)	section C.8
9	State machine		Wohrer and Zdun (2018a)	section C.9
10	String equality comparison		Volland (2019)	section C.10
11	Tight variable packing	Variables packing.	Marchesi et al. (2020)	section C.11
12	Token	Tokenisation.	(Bartoletti and Pompianu 2017)	section C.12

3.6 Gas Economic Patterns

Design patterns that aim to save gas in transactions executed through smart contracts are considered in this work as a subcategory of patterns that belongs to the **control** category. As the analysis and evaluation of gas consumption play an important role in this work, this category was considered to which the patterns **memory array building**, **string equality comparison** and **tight variable packing** belong.

As with security patterns, there are also results that are to be expected through the use of these gas economic patterns. Here it is expected that the implementation of this type of patterns reduce gas consumption in modified contracts (detailed and described in subsection 7.3.2).

Finally in this chapter, these thirty design patterns were the starting point for the search for design patterns and more patterns were discovered in the next phases of this work (see section 6.2.2).

Chapter 4

Value Analysis

In this chapter, a value analysis on this dissertation subject is presented in order to demonstrate how Solidity design patterns are related to three value concepts: value itself, perceived value and customer value. In addition, some topics about business and innovation process are explained, the current market valuation of Bitcoin, Ethereum and other main blockchain projects are also described.

4.1 Business and Innovation Process

According to Koen (2004) (Figure 4.1), the innovation process is divided into three elements: Fuzzy Front End (FFE), New Product Development (NPD) and Commercialization.

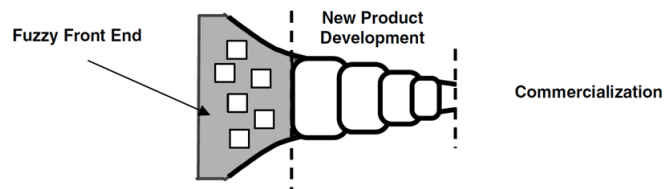


Figure 4.1: The innovation process. From Koen (2004).

- **FFE:** activities that come before the formal and well structured NPD.
- **NPD:** designed with a shaped and ordered group of tasks and questions.
- **Commercialization:** innovation process final step.

To understand the FFE better it is important to know the New Concept Development Model (NCD) which, "... provides a common language and definition of the key components of the Front End of Innovation." (Koen 2004). As can be seen in Figure 4.2, NCD is divided into three components (Koen 2004):

- The engine is where values such as leadership, culture and business strategy drive the five key elements that can be controlled by the organization of the innovation process;
- The circle includes the five elements of controllable activities: opportunity identification, opportunity analysis, idea generation, idea selection and concept definition;
- The factors influence the organizational activities until the commercialization phase. These factors consist of an outside world like the channels of distribution, law and government policy.

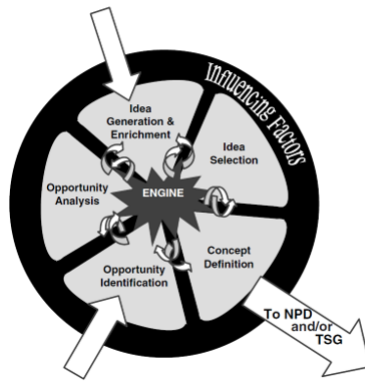


Figure 4.2: New concept development model. From Koen 2004.

The five frontend elements of NCD are applied to this dissertation theme, as follows:

1. **Opportunity identification:** the current episodes of security failures that led to hacks are a problem that can be seen as an opportunity to get better on this field and take advantage of better design patterns (detailed and described in section 1.2);
2. **Opportunity analysis:** the problem that justifies this dissertation was a result of an opportunity analysis (detailed and described in chapter 5) ;
3. **Idea genesis:** the original idea for this project was to elaborate on new methodologies that allow better development of smart contracts;
4. **Idea selection:** the decision-making process for this idea selection was based on concerns related to bad practices on smart contract development;
5. **Concept definition:** it is all the work to be produced on this work.

4.2 Value, Customer Value and Perceived Value

The concept of value is described as "... in different theoretical contexts as need, desire, interest, standard criteria, beliefs, attitudes and preferences" (Nicola, E. P. Ferreira, and J. P. Ferreira 2012). Therefore, it is not easy to find a simple definition to explain what the concept of value is. It is a relationship between the customer and the supplier that is linked to a product or service. To be able to sell anything, the vendor has to define a strategy that provides the client with advantages when purchasing the product or service (Woodall 2003). Several factors must be taken into account to define a strategy. Some of the most important are the differentiation of the competition and what is the target segment of the product, among many others. The former, because if there is no differentiation from the competition then, there is no reason for which customers have preference over a particular product or service. The latter because the more targeted the target audience is, the easier it will be to customize the product or service to meet target customer preferences (Woodall 2003).

4.2.1 Customer Value

According to Woodall (2003), "... the term customer value, is used within the marketing literature to portray both what is derived by the customer from the supplier and also what is derived by the supplier from the customer." When a customer purchases a good or service, there is always the first opinion about it, which may mean that the value of that good or service to the customer is different from the value given by the supplier. In the case of this dissertation the customers are the Solidity developers, so it is up to the author to develop a strategy (see section 6.1) so these customers can benefit from methodologies and guidelines that are recommended (detailed and described in section 8.2) here about Solidity design patterns.

4.2.2 Perceived Value

The perceived is defined as "... the consumers' overall assessment of the utility of a product based on perceptions of what is received and what is given" (Zeithaml 1988). In other words, the perceived value of a product or service is defined by the client's needs or expectations, and not for its monetary value (Woodall 2003) (Table 4.1).

Table 4.1: Benefits and sacrifices of perceived value. From Woodall (2003).

Benefits			Sacrifices
#	Attributes	Outcomes	
1	Perceived Quality	Functional Benefits	Price
2	Product Quality	Utility	Market Price
3	Quality	Use Function	Monetary Costs
4	Service Quality	Aesthetic Function	Financial
5	Technical Quality	Operational Benefits	Costs
6	Functional Quality	Economy	Costs Of Use
7	Performance Quality	Logistical Benefits	Perceived Costs
8	Service Performance	Product Benefits	Search Costs
9	Service	Strategic Benefits	Acquisition Costs
10	Service Support	Financial Benefits	Opportunity Costs
11	Special Service Aspects	Results For The Customer	Delivery and Installation Costs
12	Additional Services	Social Benefits	Costs Of Repair
13	Core Solution	Security	Training and Maintenance Costs
14	Customization	Convenience	Non-monetary Costs
15	Reliability	Enjoyment	Non-financial Costs
16	Product Characteristics	Appreciation From Users	Relationship Costs
17	Product Attributes	Knowledge, Humour	Psychological Costs
18	Features	Self-expression	Time
19	Performance	Personal Benefits	Human Energy
20		Association With Social	Groups Efforts
21		Affective Arousal	

Benefits and sacrifices related to perceived value are multiple, which is derived from different perspectives from clients and businessmen. According to Lindgreen and Wynstra (2005), "... value, as perceived by the producer, means something different from the value perceived by the user, i.e.: the producer is less sensitive to price, whereas the consumer is more sensitive to the product quality."

4.2.3 Benefits and Sacrifices

Blockchain, smart contracts and dapps have the following benefits for the customer:

- **Autonomy:** smart contracts give the customer autonomy because they eliminate intermediaries (detailed and described in subsection 2.2.4);
- **Reliability:** the intensive use of cryptography by the blockchain increases the reliability of smart contracts and dapps;
- **Security:** the distributed nature of blockchain replicates smart contract data across multiple network nodes (detailed and described in section 2.1);
- **Accuracy:** the elimination of middlemen also increases the accuracy of smart contracts.

As for the sacrifices of the customer, the following aspects are important:

- **Manpower:** need more skills and know-how (see subsection 2.1.3);
- **Costs:** in some cases, it is expensive when compared with centralized systems;
- **Popularity:** technologies are still a bit unknown to the general population.

4.3 Current Market Valuation

Blockchain and smart contract technologies are considered revolutionary because of their high level of disruption and disintermediation which can be applied in several areas (D. Tapscott and A. Tapscott 2018). In this section it is shown how the two most famous blockchains, Bitcoin and Ethereum, are innovating in a specific area of business that is finance or in the world of blockchain, the named Decentralized Finance (DeFi), through their ecosystems.

4.3.1 Bitcoin DeFi Ecosystem

Although it is also possible to develop smart contracts through tools like RSK¹ with Bitcoin blockchain, this feature is not so popular as in Ethereum. Nevertheless, there are several types of DeFi startups built on top of Bitcoin. The main application areas are²:

- **Payment processors:** payment services for merchants. Example: BitPay.³;
- **Remittance:** global transfer of money across borders. Example: Abra.⁴;
- **Lending:** service to facilitate peer-to-peer lending. Example: CoinLoan.⁵;
- **Marketplaces:** buying and selling of physical or digital products. Example: Open-Bazaar ⁶.

As for remittance, bitcoin systems allow transferring huge amounts of monetary value in exchange of incredibly reduced fees, contrarily to conventional remittance systems, such as

¹<https://www.rsk.co/>

²<https://defiprime.com/bitcoin>

³<https://bitpay.com>

⁴<https://www.abra.com/>

⁵<https://coinloan.io>

⁶<https://openbazaar.org/>

bank transfers. An example of that was a transfer of 94504 bitcoins (the equivalent to 1 billion USD at that time) charged with a fee of around 0.065 bitcoins (the equivalent to around 700 USD or to 0.007%). This transfer occurred on September 6, 2019 and can be checked here.⁷

4.3.2 Ethereum DeFi Ecosystem

In the case of Ethereum blockchain, a native platform of smart contract development, the concept of DeFi is also relevant as in Bitcoin. Moreover, DeFi in Ethereum achieved a bigger dimension because an entirely new digital Finance is being built with Ethereum smart contracts as can be observed in the following list of some of the most famous Ethereum DeFi projects:

- **MakerDAO**⁸: loans platform for collateralized debts in ether cryptocurrency;
- **Compound**⁹: autonomous protocol built for loan suppliers who gain interest in cryptocurrencies;
- **Pooltogether**¹⁰: no-loss lottery which works by pooling interest from people who contributes. The winner receives all the earned interest. Everyone else gets their money back;
- **Chainlink**¹¹: decentralized oracle data platform. These oracles are data suppliers to smart contracts.

4.3.3 Market Valuation

Market valuation is easy to execute because usually since every blockchain project has a cryptocurrency attached, whose value is based on the economic concepts of supply and demand. Table 4.2 shows the current market capitalization of Bitcoin, Ethereum and other four popular (briefly explained in subsection 2.2.7) smart contracts blockchains, namely: EOS, Cardano, Neo and Stellar.

Table 4.2: Market capitalization. Adapted from CoinMarketCap.com (2020).

#	Cryptocurrency	Market Capitalization (Euros)	Circulating Supply
1	Bitcoin	121,401,877,961	18,129,437
2	Ethereum	13,187,977,561	109,071,470
3	EOS	2,293,655,615	946,440,278
4	Neo	821,410,903	66,682,072,191
5	Cardano	805,005,696	25,927,070,538
6	Stellar	853,911,093	19,974,801,302
7	Total	139,363,838,829	

As Table 4.2 shows blockchains market is already billionaire (more than 139 billion euros). To conclude, it should be also noted that Ethereum smart contracts are an integrated part of a new series of possibilities in decentralized finance. So, to prevent monetary losses due

⁷<https://www.blockchain.com/btc/address/37XuVSEpWW4trkfmvWzegTHQt7BdktSKUs?page=5>

⁸<https://makerdao.com>

⁹<https://compound.finance/>

¹⁰<https://www.pooltogether.com/>

¹¹<https://chain.link/>

to security failures and to avoid hacks, it is fundamental that design patterns on Solidity smart contracts to be solid and well-identified (see section 3.1).

4.4 The Analytic Hierarchy Process

Nowadays, the use of frameworks enables a faster and more organized process of software development is prevalent. Particularly, in the case of Solidity smart contracts the market already offers several frameworks that have the goal to make the construction of smart contracts and dapps easier. This section relies on the Analytic Hierarchy Process (AHP), in order to choose the most suitable framework to be used as a tool by the author in the context of this dissertation.

AHP features a measurement or classification theory based on pairwise comparisons of alternatives. This technique is based on a scale of priorities defined according to the judgments of experts in the area of the problem. These judgments can be inconsistent. However, the AHP method allows both to quantify this consistency and to improve such judgments (Saaty 2008).

According to Saaty (2008), in order to make a decision based on the AHP, it is necessary to follow these stages:

1. Define the problem and determine the type of knowledge required;
2. Structure of the decision hierarchy from top to bottom, specifying the main objective, criteria and alternatives;
3. Build a set of comparison matrices pair by pair, considering the different levels of hierarchy;
4. Use the priorities obtained through the level comparisons to generate the global priority of the alternatives.

The following criteria were defined as fundamental for the decision on the framework of a smart contract:

- **Learning Curve:** the expected amount of time for the author to get comfortable with the usage of a framework;
- **Market Share:** the popularity level of a framework among communities of smart contracts development;
- **Features:** functionalities that can be advantages on the usage of a framework by the author;
- **Documentation:** quality, quantity, organization and structure of official support documents of a framework.

The three alternative frameworks are:

- **Embark**¹²
- **Waffle**¹³
- **Truffle**¹⁴

The hierarchical diagram is represented in Figure 4.3.

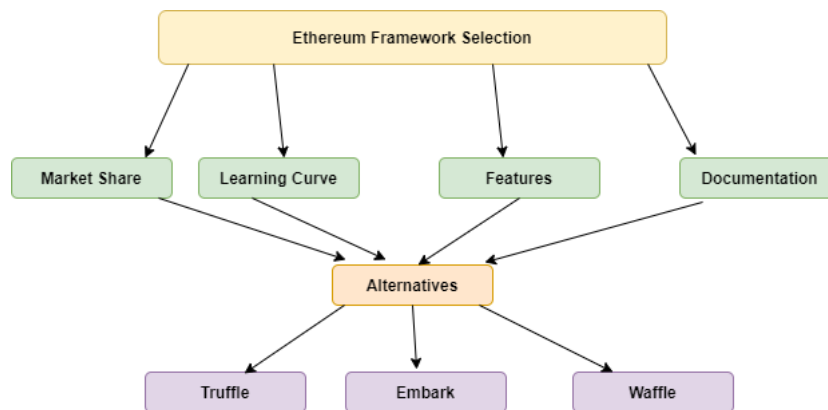


Figure 4.3: AHP diagram.

The AHP fundamental scale is utilized to designate weights, (see Table 4.3).

Table 4.3: The AHP fundamental scale. From Saaty (2008).

Intensity	Definition
1	Equal Importance
2	Weak or slight
3	Moderate importance
4	Moderate plus
5	Strong importance
6	Strong plus
7	Very strong or demonstrated importance
8	Very, very strong
9	Extreme importance

Table 4.4 shows the relative importance of each of the selection criteria, comparing each pair of objectives and assigning a classification, based on Table 4.3 fundamental scale.

The defined classifications seek to reflect the difficulties and requirements of the author regarding the difficulties he expects to encounter and support the tasks he intends to perform.

¹²<https://framework.embarklabs.io/>

¹³<https://getwaffle.io/>

¹⁴<https://www.trufflesuite.com/>

Table 4.4: AHP criteria comparison.

	Learning Curve	Market Share	Documentation	Features
Learning Curve	1	9	5	2
Market Share	1/9	1	1/5	1/7
Documentation	1/5	5	1	1/2
Features	1/2	7	2	1

Learning Curve

The learning curve is the main decision criterion because technical knowledge is fundamental to the success of this dissertation. Table 4.5 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.5: AHP comparison for learning curve.

Learning Curve	Truffle	Embark	Waffle
Truffle	1	4	1/2
Embark	1/4	1	1/5
Waffle	2	5	1

Features

The features criterion is important to distinguish which frameworks have the most useful features for the development of this dissertation. Table 4.6 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.6: AHP comparison for features.

Features	Truffle	Embark	Waffle
Truffle	1	1/2	2
Embark	2	1	4
Waffle	1/2	1/4	1

Documentation

Documentation is fundamental but less important than the learning curve and features criteria for the framework selection. Table 4.7 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.7: AHP comparison for documentation.

Documentation	Truffle	Embark	Waffle
Truffle	1	1/2	1/5
Embark	2	1	1/3
Waffle	5	3	1

Market Share

The market share criterion seeks to reflect the importance of Truffle, Embark and Waffle in the market of Solidity frameworks. Table 4.8 presents the pairwise comparison between the different alternatives for this criterion.

Table 4.8: AHP comparison for market share.

Market Share	Truffle	Embark	Waffle
Truffle	1	7	4
Embark	1/7	1	1/5
Waffle	1/5	5	1

Results

To decide which framework is the best option, an AHP tool¹⁵ (developed in R language) was used to calculate all the previous weight tables, as can be seen in Figure 4.4

	Weight	Truffle	Embark	Waffle	Inconsistency
Selecting the best Ethereum framework	100.0%	45.0%	30.4%	24.6%	3.4%
Learning Curve	53.6%	30.5%	17.9%	5.2%	2.3%
Features	27.6%	3.9%	7.9%	15.8%	0.0%
Documentation	14.6%	9.5%	1.8%	3.4%	0.4%
Market Share	4.2%	1.0%	2.9%	0.3%	11.8%

Figure 4.4: The AHP results.

Truffle had the best rating with 45%, Embark in second place with a rating of 30.4% and in the last Waffle with 24.6%.

¹⁵<https://github.com/gluc/ahp/>

Chapter 5

Problem Statement

After presented and explored several topics, this chapter intends to detail the problem and the objectives that were briefly presented in the first chapter of this document, namely in section 1.2 and section 1.3.

5.1 Main Issues

The main issues concerning the development of Solidity smart contracts are related to the four categories of design patterns found in section 3.1: Security, Authorization, Control and Maintenance.

In the specific case of this work, it is intended to understand how the use of certain design patterns affects gas consumption and the number of vulnerabilities in selected contracts of all categories before mentioned.

Security

This category of patterns has great importance, as most of Ethereum smart contracts are programmed to execute the transference of monetary funds, in the form of crypto-assets (Bartoletti and Pompianu 2017).

Despite Ethereum is yet a very recent platform (started in 2014) it has already a considerable history of hacks which involved the theft of big amounts of money (Alchemy 2018). Furthermore, there is also a known list of vulnerabilities on Ethereum Smart Contracts. Table 5.1 shows a summary of those vulnerabilities (taxonomy from DASP (2020)):

Table 5.1: Ethereum vulnerabilities. From Durieux et al. (2019)

#	Name	Description
1	Access Control	Failure to use function modifiers or use of tx.origin.
2	Arithmetic	Integer over/underflows.
3	Bad Randomness	Malicious miner biases the outcome.
4	Denial of service	The contract is overwhelmed with time-consuming computations.
5	Front running	Two dependent transactions invoking same contract are included in one block.
6	Reentrancy	Reentrant function calls make a contract to behave in an unexpected way.
7	Short addresses	EVM itself accepts incorrectly padded arguments.
8	Time manipulation	The timestamp of the block is manipulated by the miner.
9	Unchecked low level	call(), callcode(), delegatecall() or send() fails and it is not 10 checked calls.
10	Unknown Unknowns	Vulnerabilities not identified in DASP (2020).

As this work is not only about Solidity security issues, the other three categories of design patterns are explained as well. Table 5.2 shows a summary of examples of tasks related to each category.

Table 5.2: Design patterns categories.

#	Category	Type of tasks	Examples
1	Authorization	User permission tasks	Restrict some features of a contract only to its owner address.
2			Only allow the execution of a transaction after being authorized by a group of users.
3			Only allow the execution of a transaction after some kind of authentication.
4	Control	Operational tasks	Smart contract interactions with off-chain data.
5			Manage some data of a smart contract with confidentiality.
6			Standard functions related to crypto-assets (ERC-20 tokens).
7	Maintenance	Life cycle tasks	Destruction/inactivation of smart contracts.
8			Automatic creation of smart contracts.
9			Add new features to a smart contract.

5.2 Objectives

The objectives of this dissertation were already defined in section 1.3, but in this section they are related to several tasks.

Table 5.3 shows the tasks for the first objective (**Collect source code of prominent Solidity contracts, following criteria defined in subsection 6.1.2, through a blockchain explorer tool (see section 2.2.6) and analyze pattern usage**).

Table 5.3: Tasks related to first objective.

#	Task
1	Make an exhaustive study of design patterns on Ethereum/Solidity smart contracts (see section 3.1).
2	Define the criteria for collecting a dataset of smart contracts through a blockchain explorer (see subsection 6.1.2).
3	Compare pattern usage in the dataset with the ones selected from section 3.1 (see section 6.2.2).

Table 5.4 shows the tasks for the second objective (**Analyze the existence of security vulnerabilities on selected smart contracts (see section 2.2.6), explore their pattern usage and gas consumption**).

Table 5.4: Tasks related to second objective.

#	Task
1	Choose the security tools to use (see section 2.2.6).
2	Statistically analyze vulnerabilities in the contracts of the dataset (see section 6.3).
3	Compare the original contracts and their modified version (with the implementation of design patterns), concerning the number of vulnerabilities and gas usage (see section 7.2 and section 7.3).

Table 5.5 shows the tasks for the third objective (**Examine the effect of some design patterns on modified smart contracts**).

Table 5.5: Tasks related to third objective.

#	Task
1	Analyze the results obtained in the two previous objectives.
2	Understand how to take advantage of design patterns in terms of security and gas consumption.
3	Define recommendations related to, specifically, security and gas usage and generically to smart contracts development (see section 8.2).

Chapter 6

Design and Implementation

This chapter presents the practical component of the project by explaining its main development phases. Firstly, the design phase is detailed, by describing what tools and design patterns were chosen. Secondly, it demonstrates how the smart contracts collection from a blockchain explorer tool was conducted. Thirdly, it introduces two different approaches used on the search for Solidity design patterns. Finally, results are shown and several design patterns implementations are given as examples in collected contracts that are suitable for that purpose.

6.1 Design

6.1.1 Selection of Technologies and Patterns

This phase was essentially a choice of tools and design patterns to make the implementation of the practical and technical part of this work possible. These choices are presented below:

- Blockchain explorer - **Etherscan**: three options were studied in section 2.2.6. The choice fell on Etherscan because it has the most suitable API for this project, allowing the download of the code of the smart contracts and some data provided, such as the number of transactions each contract carried out in a given period;
- Security tools - **Mythril** and **Slither**: Given the results of these two security tools in the study carried out by Durieux et al. (2019) (see section 2.2.6), they were chosen as an integral part of this project;
- Solidity Integrated Development Environment (IDE) - **Remix**¹: This compiler was chosen because it allows to easily calculate gas consumption in Solidity contracts;
- Solidity framework - **Truffle**: This framework for the development of Solidity smart contracts was selected on the basis described in section 4.4;
- Statistical tool - **R language**² and **RStudio**³: These statistical tools were chosen because of the author's previous experience;
- Scripting language - **JavaScript/Node.js**⁴: This scripting language and run time platform were chosen because of the author's previous experience;

¹<https://remix.ethereum.org/>

²<https://www.r-project.org/>

³<https://rstudio.com/>

⁴<https://nodejs.org/>

- **Design patterns:** the patterns selected in section 3.2, section 3.3, section 3.5 and section 3.4 where it was described why they were chosen.

6.1.2 Detailed Approach

The following approach follows the methodology delineated in section 1.3 and was defined to accomplish the goals of this work, by encompassing the following nine phases:

1. Use a blockchain explorer to collect the Ethereum contracts that follow these criteria:
 - They have recent transactions (between March 1, 2020 and March 31, 2020);
 - They have one hundred transactions or more on the previous period;
 - They are marked as “verified” by the blockchain explorer.
2. Look for design patterns used in the dataset of contracts downloaded in the previous step;
3. Compare pattern usage of the dataset with the ones found in section 3.1;
4. Statistically analyze the patterns usage in the contracts of the dataset;
5. Apply security tools described in section 2.2.6 in the contracts of the dataset;
6. Statistically analyze vulnerabilities in the contracts of the dataset;
7. Apply design patterns in original contracts of the dataset;
8. Compare gas usage and number of vulnerabilities between original contracts and their modified version with design patterns implementation;
9. Make recommendations based on the results of this work.

6.2 Collecting Smart Contracts

The dataset for this project included smart contracts collected through the Etherscan API. Two Node.js scripts were developed. The first⁵ one gets contract’s addresses that fulfill the requirements defined in subsection 6.1.2. Figure 6.1 delineates its algorithm (the code is in section E.2).

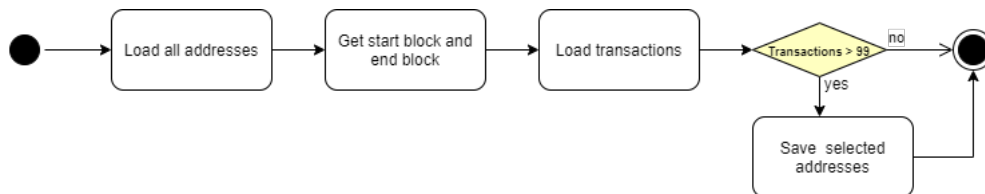


Figure 6.1: Get selected addresses algorithm.

⁵<https://github.com/jmesmoorish/tdmei/blob/master/getAddresses.js>

The purpose of the second script⁶ was to download the Solidity code of the contracts through the addresses found on the first script. Three hundred and sixty smart contracts were automatically obtained as a result of these two scripts. Figure 6.2 shows the algorithm of second Node.js script (the code is in section E.3).

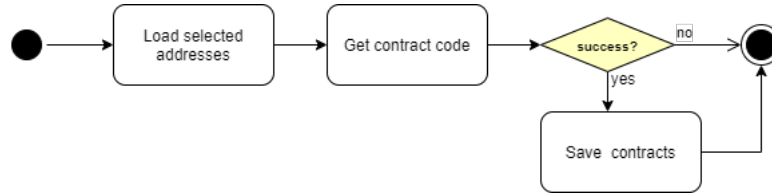


Figure 6.2: Get selected contracts algorithm.

In order to detect possible duplicate contracts, it was necessary to apply an R programmed script⁷ (see the code in section E.4). Hence forty-two duplicated contracts were found that were excluded from the dataset. The baseline for this project was a dataset of three hundred and eighteen smart contracts, which were carefully scrutinized for Solidity design patterns.

Figure 6.3 shows a histogram that classifies the contracts dataset by the number of transactions performed, at intervals of 2000, during the period between March 1, 2020 and March 31, 2020.

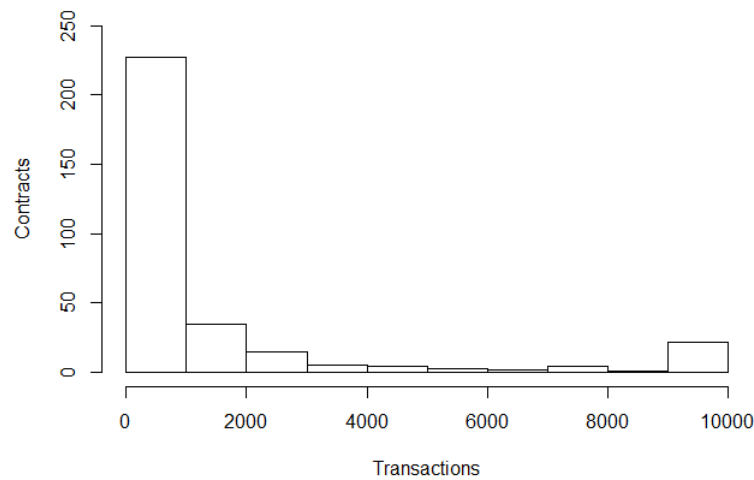


Figure 6.3: Transactions histogram.

As can be seen in Figure 6.3, the large majority of the contracts had less than one thousand transactions in the period defined on this project to collect only contracts with recent activity.

⁶<https://github.com/jmesmoorish/tdmei/blob/master/getContracts.js>

⁷<https://github.com/jmesmoorish/tdmei/blob/master/r-code/dupe-files.R>

Table 6.1 shows the eighteen collected smart contracts with more than ten thousand transactions in the period between March 1, 2020 and March 31, 2020. It should be noted that Ethercan API only retrieves at most the last ten thousand transactions of a contract in a determined period.

Table 6.1: Collected contracts with more transactions.

#	Contract	Address
1	BrokerV2	0x7ee7ca6e75de79e618e88bdf80d0b1db136b22d0
2	CapitalCoinExtended	0xacde447a4f3516b732c06b715044e528475ae1b4
3	Dai	0x6b175474e89094c44da98b954eedeac495271d0f
4	Exchange	0xaA5bBD5A177A588b9F213505cA3740b444Dbd586
5	Flipper	0xd8a04f5412223f513dc55f839574430f5ec15531
6	HBTOKEN	0x1ecb25dffce3cd2824b446e2e7e9d3f48b6f9c90
7	LUCKYToken	0xE478d4F4a87D4D641AF97ca0b5Cc3dB61e266357
8	MainContract	0x4c717a5fa94adce745dbdd1c485c0666d2656444
9	MRDF	0x5f3e90b7ecfbc4f63b60c55ec0e825a927467bcc
10	Oracle	0x64fe692be4b42f4ac9d4617ab824e088350c11c2
11	ProofOfContributionChain	0x912656188616e0184e3181f019022990a63280b1
12	RHC	0x435d4183ae0aeb1babc31bd25a815d6244fc3562
13	S3Coin	0x2c702d98d4d7e2af742f071c2038565b1cc07bef
14	SBToken	0xcadb96858fe496bb6309622f9023ba2defb5d540
15	SignedOperationProxy	0x2a842bc64343fad4ec4a8424ba7ff3c0a70b6e55
16	TEPTOKENV2	0xd7cc16500d0b0ac3d0ba156a584865a43b0b0050
17	TokenProxy	0x1dd0df760eb950083c1925da19fc7ac1356a190b
18	TrueProfileLogic	0xd0466448fb6bf17beaf9325974b7f71c6937609b

6.2.1 Text Mining Search Approach

The first approach for design patterns discovery in the contracts dataset was to use automated tools to accelerate this process. For that purpose, an R script⁸ was developed, which used a text mining R package - Quanteda⁹. The main idea of this script is divided into three phases. The first phase is to load all 318 contracts in memory, the second phase removes all lexical Solidity words from contracts, and the third phase looks for patterns through a dictionary, where are associated common identifications to each pattern, found on literature. Figure 6.4 shows the algorithm of this R script (the code is in section E.1).

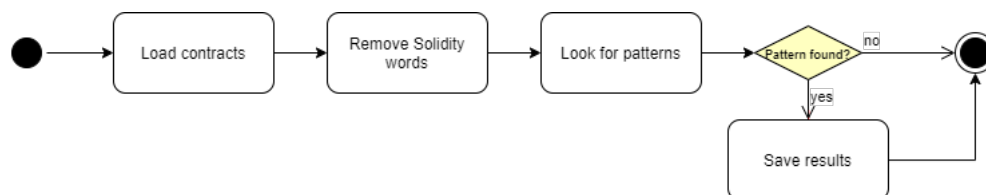


Figure 6.4: Text mining algorithm.

⁸<https://github.com/jmesmoorish/tdmei/blob/master/r-code/quanteda.R/>

⁹<https://quanteda.io/>

Table 6.2 shows the results of this approach. The contracts column shows how many contracts the identification of a pattern was found in, and the total column shows how many times those identifiers were found.

Table 6.2: Text mining search results.

#	Category	Pattern	Contracts	Total
1	Authorization	Access restriction	0	0
2		Multiple authorization	0	0
3		Ownership	217	1323
4	Control	Commit and reveal	0	0
5		Guard check	0	0
6		Memory array building	0	0
7		Oracle	18	290
8		Poll	1	54
9		Pull payment	0	0
10		Randomness	8	66
11		Safemath	244	1240
12		State machine	1	1
13		String equality comparison	0	0
14		Tight variable packing	0	0
15		Token	228	3942
16	Maintenance	Automatic deprecation	0	0
17		Contract composer	0	0
18		Contract factory	0	0
19		Contract register	0	0
20		Contract relay	31	455
21		Data segregation	0	0
22		Mortal	0	0
23		Satellite	0	0
24	Security	Balance limit	0	0
25		Checks-effects-interaction	0	0
26		Emergency stop	40	166
27		Mutex	14	20
28		Rate limit	0	0
29		Secure ether transfer	0	0
30		Speed bump	0	0

Unfortunately, this approach proved to have low accuracy because the identification and names given to design patterns by programmers differ from those that are found in the literature. Another problem is that patterns like **checks-effects-interaction**, **tight variable packing** and **memory array building** are never identified by any name because their implementations depends on uniquely in code reordering or code optimizations. Nevertheless, this approach was important in the cases where it was accurate, for familiarization with the way how Solidity programmers usually apply some of the design patterns which were described in chapter 3.

6.2.2 Manual Search Approach

Due to the fact that the text mining approach was not precise enough, it was necessary to execute an exhaustive manual search for Solidity design patterns.

The 318 smart contracts were read by the author file by file, line by line. A spreadsheet file for each contract was created, in which the design patterns used and their location (starting line in the code) were recorded. An example is shown in figure 6.5.

	A	B
1	Pattern	Line
2	Ownership	38
3	Safemath	3
4	Token	64
5	checks-effects-interaction	143
6	checks-effects-interaction	167
7	Pull Payment	200

Figure 6.5: Spreadsheet for registering of found design patterns.

To count how many times each design pattern was found, it was necessary to apply an Node.js programmed script¹⁰ which allowed the automatic elaboration of manual search results, that are shown in the next sections.

Known Design Patterns Found

All the design patterns selected from chapter 3 were found at least one time in this manual search for design patterns. Table 6.3 shows the results of this approach. The contracts column shows how many contracts the identification of a pattern was found in, and the total column shows how many times those identifiers were found.

¹⁰<https://github.com/jmesmoorish/tdmei/blob/master/getPatternsCount.js/>

Table 6.3: Manual search results - known patterns.

#	Category	Pattern	Contracts	Total
1	Authorization	Access restriction	97	241
2		Multiple authorization	5	5
3		Ownership	206	206
4	Control	Commit and reveal	2	2
5		Guard check	21	72
6		Memory array building	10	13
7		Oracle	17	17
8		Poll	3	3
9		Pull payment	52	52
10		Randomness	4	4
11		Safemath	253	253
12		State machine	7	7
13		String equality comparison	1	1
14		Tight variable packing	75	118
15		Token	209	212
16	Maintenance	Automatic deprecation	14	14
17		Contract composer	137	253
18		Contract factory	11	13
19		Contract register	8	9
20		Contract relay	29	29
21		Data segregation	19	33
22		Mortal	22	22
23		Satellite	6	6
24	Security	Balance limit	9	9
25		Checks-effects-interaction	177	541
26		Emergency stop	52	52
27		Mutex	22	22
28		Rate limit	12	12
29		Secure ether transfer	7	7
30		Speed bump	14	18

The design pattern found in more contracts was **safemath**, while the pattern which was utilized more times was **checks-effects-interaction**. On the contrary, **string equality comparison** was the least used pattern.

Table 6.4 shows how many design patterns implementations were found in the dataset of contracts divided by intervals of transaction activity. This calculation was done through this Node.js script¹¹.

¹¹<https://github.com/jmesmoorish/tdmei/blob/master/getPatternsCountByTxCount.js/>

Table 6.4: Pattern implementations by contracts transactions (thousands).

#	Category	pattern	[0-2]]2-4]]4-6]]6-8]]8-10]
1	Authorization	Access restriction	194	25	2	5	14
2		Multiple authorization	4	1	0	0	0
3		Ownership	171	14	5	5	11
4	Control	Commit and reveal	2	0	0	0	0
5		Guard check	62	0	0	0	10
6		Memory array building	11	0	0	0	2
7		Oracle	14	0	0	1	2
8		Poll	2	1	0	0	0
9		Pull payment	45	4	1	1	1
10		Randomness	4	0	0	0	0
11		Safemath	207	19	6	4	17
12		State machine	7	0	0	0	0
13		String equality comparison	1	0	0	0	0
14		Tight variable packing	99	8	0	2	9
15		Token	172	15	6	4	15
16	Maintenance	Automatic deprecation	12	1	0	0	1
17		Contract composer	200	16	5	3	11
18		Contract factory	6	2	0	1	0
19		Contract register	6	2	0	1	0
20		Contract relay	11	1	0	1	0
21		Data segregation	28	2	0	1	2
22		Mortal	21	1	0	0	0
23		Satellite	5	1	0	0	0
24	Security	Balance limit	6	0	0	1	2
25		Checks-effects-interaction	371	30	14	6	30
26		Emergency stop	41	5	2	2	2
27		Mutex	20	0	0	0	2
28		Rate limit	10	1	0	0	1
29		Secure ether transfer	6	0	0	1	0
30		Speed bump	14	3	0	0	1

As can be seen in Table 6.4 more frequent patterns have the biggest quantity of implementations in all intervals of transactions.

Unknown Design Patterns Found

One of the tasks defined to the search for design patterns in the collected contracts was to check if the patterns found matched the patterns described in chapter 3. However, four patterns (**Address**¹², **Byteslib**¹³, **Context**¹⁴ and **Roles**¹⁵) that are not usually described in recent literature but can be found on code repositories like Github, were found. Table 6.5 shows the unknown design patterns found.

¹²<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/Address.sol/>

¹³<https://github.com/GNSPS/solidity-bytes-utils/blob/master/contracts/BytesLib.sol/>

¹⁴<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/GSN/Context.sol/>

¹⁵<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/access/AccessControl.sol/>

Table 6.5: Manual search results - unknown patterns.

#	Pattern	Contracts	Total	Category	Sample
1	Roles	33	33	Authorization	section A.4
2	Address	35	35	Control	section C.13
3	Byteslib	6	6	Control	section C.14
4	Context	38	38	Control	section C.15

Table 6.6 describes the four unknown design patterns found in a similar way of chapter 3.

Table 6.6: Unknown patterns - description.

#	Pattern	Category	Problem	Solution
1	Roles	Authorization	Any party can call any contract function.	Restrict specific sensitive tasks only to users with certain roles.
2	Address	Control	Operations with the address type variables are frequent in Solidity.	Create a compilation of utility functions to manage variables of the address type.
3	Byteslib	Control	Operations with the bytes type variables are frequent in Solidity.	Create a compilation of utility functions to manage variables of the bytes type.
4	Context	Control	Checking the current execution context in a contract is a frequent operation.	Create reusable methods related to smart contracts context.

Smart Contracts Taxonomy

The 318 collected contracts were divided into four different types, namely: token, DeFi, game and other (which the autor could not identify concretely the type). Table 6.7 shows how many smart contracts per type were found.

Table 6.7: Collected smart contracts taxonomy.

#	Type	Total
1	Token	202
2	DeFi	68
3	Game	7
4	Other	41

The fact that about 65.52% of the collected contracts pertains to the token type is significant because this contract type usually has a pattern with the same name (**token**) implemented.

Table 6.8 shows how many design patterns implementations were found by smart contract type. This calculation was done through this Node.js script¹⁶.

Table 6.8: Pattern implementations by contract type.

#	Category	Pattern	Token	DeFi	Game	Other
1	Authorization	Access restriction	93	98	10	40
2		Multiple authorization	3	0	0	2
3		Ownership	130	49	5	22
4	Control	Commit and reveal	0	1	0	1
5		Guard check	22	40	9	1
6		Memory array building	4	8	0	1
7		Oracle	1	13	1	2
8		Poll	1	0	0	2
9		Pull payment	32	13	4	3
10		Randomness	1	0	2	1
11		Safemath	176	45	5	27
12		State machine	1	4	1	1
13		String equality comparison	0	1	0	0
14		Tight variable packing	27	56	11	24
15		Token	202	3	2	5
16	Maintenance	Automatic deprecation	4	6	0	4
17		Contract composer	123	80	6	26
18		Contract factory	1	5	2	5
19		Contract register	0	4	0	5
20		Contract relay	4	14	0	11
21		Data segregation	3	17	1	12
22		Mortal	7	8	4	3
23		Satellite	1	2	0	3
24	Security	Balance limit	2	5	0	2
25		Checks-effects-interaction	423	11	10	7
26		Emergency stop	41	10	0	1
27		Mutex	6	15	0	1
28		Rate limit	5	7	0	0
29		Secure ether transfer	5	1	0	1
30		Speed bump	7	2	0	9

Once more in token type the **checks-effects-interaction** can be highlighted with 423 implementations found. In DeFi type, **access restriction** and **contract composer** are the most frequent patterns. As far as game type is concerned, **access restriction**, **tight variable packing** and **checks-effects-interaction** have similar numbers (around 10 implementations). Finally, in the other type category **access restriction** is at the top and **safemath**, **tight variable packing** and **contract composer** are the following patterns with more frequency.

¹⁶<https://github.com/jmesmoorish/tdmei/blob/master/getPatternsCountByType.js/>

6.3 Security Analysis

6.3.1 Mythril

The static security tool Mythril has revealed to be heavy with very long times of execution for each contract file (some take more than ten minutes). Another problem with this tool was the number of contracts that could not be analyzed because the analysis could not be completed due to errors.

Two Node.js scripts were also developed here to automate two different tasks. The first¹⁷ was to run all the 318 analyses (one for each collected contract file) without being necessary to run a command line on the console for each contract. The second¹⁸ was to gather and organize the results that can be seen in Table 6.9 which shows vulnerabilities found by Mythril analysis.

Table 6.9: Mythril results.

#	Vulnerability	Contracts	Total
1	Assert violation	41	75
2	Reentrancy	36	58
3	Integer overflow and underflow	14	30
4	DoS with failed call	16	22
5	Block values as a proxy for time	6	12
6	Delegate call to untrusted callee	5	9
7	Weak sources of randomness from chain attributes	2	3
8	Unprotected self-destruct instruction	3	3
9	Use of deprecated solidity functions	3	3
10	Unprotected ether withdrawal	3	3
11	Unchecked call return value	1	2

Reentrancy and **assert violation** were the most frequent vulnerabilities. The smart contract with more issues is called Cube¹⁹ with 11 vulnerabilities. In 147 contracts no vulnerabilities were found in, and in 84 contracts analysis was not possible due to errors.

6.3.2 Slither

The static security tool Slither has revealed to be much lighter than Mythril, with executions times of few seconds. Unfortunately, in terms of errors, this tool has similar numbers with Mythril. Two scripts were developed, the first²⁰ to automate the analysis execution on contract files and the second²¹ to collect the results of each analysis.

¹⁷<https://github.com/jmesmoorish/tdmei/blob/master/mythril-code/mythril.js/>

¹⁸<https://github.com/jmesmoorish/tdmei/blob/master/mythril-code/getMythrilStats.js/>

¹⁹<https://etherscan.io/address/0x777777f1c2c60fbd4cfaa96d780f06afa540ee30/>

²⁰<https://github.com/jmesmoorish/tdmei/blob/master/slither-scode/slither.js/>

²¹<https://github.com/jmesmoorish/tdmei/blob/master/slither-scode/getSlitherStats.js/>

Table 6.10 shows vulnerabilities found by Slither analysis.

Table 6.10: Slither results.

#	Vulnerability	Contracts	Total
1	Shadowing-local	67	271
2	Reentrancy-events	55	196
3	Unused-return	36	106
4	Incorrect-equality	24	77
5	Reentrancy-benign	32	74
6	Calls-loop	17	72
7	Constant-function-asm	20	68
8	Uninitialized-local	26	65
9	Timestamp	23	60
10	Reentrancy-no-eth	28	49
11	Locked-ether	39	39
12	Erc20-interface	12	32
13	Arbitrary-send	18	23
14	Shadowing-state	9	18
15	Uninitialized-state	9	15
16	Controlled-delegatecall	3	6
17	Shadowing-abstract	5	5
18	Erc721-interface	1	4
19	Reentrancy-eth	2	4
20	Tx-origin	2	2
21	Shadowing-builtin	2	2
22	Void-cst	1	2
23	Suicidal	1	1

Shadowing-local and **reentrancy-events** were most frequently found vulnerabilities. The smart contract with more issues is called Aggregator ²² with 32 vulnerabilities. In 60 contracts Slither did not find any vulnerability, and in 85 contracts analysis was not possible due to errors.

6.4 Design Pattern Implementations

The final stage of this chapter consisted of implementing design patterns in some of the 318 collected contracts of the dataset. The following four sections present an example of an implementation for each design pattern category.

6.4.1 Authorization Patterns

The **access restriction** pattern was found in 97 contracts with a total of 241 implementations. Listing 6.1 shows an example of a contract that was modified to benefit from this design pattern.

²²<https://etherscan.io/address/0x560b06e8897a0e52dbd5723271886bbcc5c1f52a/>

```

1 modifier condition(bool _condition) { //access restriction pattern
2     require(_condition);
3     _;
4 }
5
6 function transferOwnership(address newOwner) public onlyOwner condition(
7     newOwner != address(0)) {
8     //require(newOwner != address(0));
9     emit LogOwnershipTransferred(owner, newOwner);
10    owner = newOwner;
11 }

```

Listing 6.1: Contract modified by access restriction pattern.

In this example, a very generic implementation of **access restriction** pattern was developed, through the creation of a modifier (line 1) which can be used to check any condition and can be applied to any function (line 6). This modification was applied to the Acid²³ smart contract.

6.4.2 Control Patterns

The **guard check** pattern was found in 21 contracts with a total of 72 implementations. Listing 6.2 shows an example of a contract that was modified to benefit from this design pattern.

```

1 function startAirdrop(uint256 _aSBlock, uint256 _aEBlock, uint256 _aAmt,
2     uint256 _aCap) public onlyOwner() {
3     require(_aEBlock != 0, 'End block cannot be 0!'); //guard check
4     require(_aEBlock >= _aSBlock, 'End block cannot be minor than start
5     block!'); //guard check pattern
6     aSBlock = _aSBlock;
7     aEBlock = _aEBlock;
8     aAmt = _aAmt;
9     aCap = _aCap;
10    aTot = 0;
11 }

```

Listing 6.2: Contract modified by access restriction pattern.

Here the AFX²⁴ smart contract is modified by applying the **guard check** pattern through two validations (lines 3 and 4) in a function, to verify two user inputs.

²³<https://etherscan.io/address/0x23ea10cc1e6ebdb499d24e45369a35f43627062f/>

²⁴<https://etherscan.io/address/0x338743f30729c237272ef36ee5fe9e0ff1adcf6f/>

6.4.3 Maintenance Patterns

The **contract factory** pattern was found in 11 contracts with a total of 13 implementations. Listing 6.3 shows an example of a contract that was modified to benefit from this design pattern.

```

1 contract CreativesStorage_Factory is Ownable { //contract factory
2     address [] public deployedCreativesStorages;
3
4     function createCreativesStorage() public {
5         address newCreativesStorage = address (new CreativesStorage ());
6         deployedCreativesStorages.push(newCreativesStorage);
7     }
8
9     function getDeployedCreativesStorage() public view returns (address
10    [] memory) {
11         return deployedCreativesStorages;
12    }
13 }
```

Listing 6.3: Contract modified by access restriction pattern.

In this example, an implementation of the **contract factory** pattern was developed, through the creation of an auxiliary contract (line 1) which has the purpose to deploy instances of CreativesStorage contract on Ethereum blockchain. This modification was applied to the CreativesRegistry²⁵ smart contract.

6.4.4 Security Patterns

The **mutex** pattern was found in 22 contracts (one implementation in each contract). Listing 6.4 shows an example of a contract that was modified to benefit from this design pattern.

```

1 modifier noReentrancy() { // mutex pattern
2     require(!locked);
3     locked = true;
4     _;
5     locked = false;
6 }
7
8 function rely(address usr) external note auth noReentrancy {
9     wards[usr] = 1;
10 }
```

Listing 6.4: Contract modified by access restriction pattern.

In this example a modification was implemented through **mutex** pattern, by creating a modifier (line 1) which can be applied to a function (line 8) to avoid the **reentrancy**²⁶ vulnerability, by locking sensitive operations with a boolean variable (locked). This modification was applied to the KriosToken²⁷ smart contract.

²⁵<https://etherscan.io/address/0xa04b819fa3b94ee8d3713adb136d9de120e01f86/>

²⁶<https://swcregistry.io/docs/SWC-107>

²⁷<https://etherscan.io/address/0x42566cfefc853c232117eba4413e45782a72715d/>

Chapter 7

Evaluation

This chapter has the purpose of evaluating some of the design patterns which were researched and selected in chapter 3 and later discovered and confirmed in the dataset of contracts retrieved from Ethereum blockchain through Etherscan, in chapter 6. On top of that, eight design patterns (two from each category) were selected in which the implementation was tested in ten different collected contracts, in terms of security failures by Mythril and Slither, and terms of gas usage by Remix IDE. Both evaluations were done by comparing results before and after implementations in the selected smart contracts.

7.1 Limitations on Selecting Patterns and Contracts

Only two design patterns were chosen per category because evaluating all thirty patterns (ideally) would make this work too time-consuming. This option may result in the results not being as accurate as would be desirable. However, evaluating these eight patterns already gives a general sense of the impact of implementing Solidity patterns in smart contracts. In this context, the selected design patterns and their justification were as follows:

- **Authorization:** **ownership** and **access restriction** patterns were chosen for this category, because they are the most frequent patterns in the contracts dataset (in **authorization** category);
- **Control:** **guard check** and **tight variable packing** patterns were chosen for this category because they are found many times in the contracts dataset (in **control** category) and their implementation has a small impact on the modifications of the contracts. **Tight variable packing** pattern was also selected because it belongs to the subcategory of gas economic patterns;
- **Maintenance:** **mortal** and **contract factory** patterns were chosen for this category because their implementation is simple and has a small impact on the modifications of the contracts;
- **Security:** **checks-effects-interaction** and **mutex** patterns were chosen for this category because they have the purpose of preventing reentrancy vulnerability, which was the security issue found more times in the combination of Mythril and Slither results.

Ten contracts were selected for each category with an additional ten contracts to implement all eight design patterns, thus totaling fifty modified contracts. This choice was almost random as the only criterion for choosing among the 318 contracts collected was that they were not too large in terms of lines of code (maximum 1000 lines), to speed up the security

evaluation mainly on Mythril (which runs much slower than Slither) and also speed up gas consumption evaluation in Remix IDE.

7.2 Security

7.2.1 Authorization Patterns

Table 7.1 shows the results.

Table 7.1: Authorization category - security evaluation.

#	Contract	Mythril before	Mythril after	Slither before	Slither after
1	Acid	0	0	3	3
2	DCEP	0	0	0	0
3	DrepToken	0	0	0	0
4	GL_ERC20	0	0	0	0
5	OffChainOrFeedPriceFeed	0	0	0	0
6	ProofOfContributionChain	0	0	0	0
7	RVC	1	1	0	0
8	TokenERC20	0	0	0	0
9	WXB	0	0	0	0
10	YSM	0	0	0	0
Total		1	1	3	3

As can be observed in Table 7.1, **ownership** and **access restriction** patterns did not have an impact on static security analysis in both tools, Mythril (one vulnerability before and after) and Slither (three vulnerabilities before and after).

Mythril detected *assert violation*¹ vulnerability in RVC contract that could be fixed simply by changing the Solidity expression *assert* for *require* expression.

In Acid contract, Slither detected *incorrect-equality*² vulnerability two times that could be fixed by changing the equality to comparison type of bigger than or smaller than.

Slither found *reentrancy* vulnerability one time in Acid contract, that could be corrected by the **mutex** or **checks-effects-interaction** patterns.

¹<https://swcregistry.io/docs/SWC-110/>

²<https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities/>

7.2.2 Control Patterns

Table 7.2 shows the results.

Table 7.2: Control category - security evaluation.

#	Contract	Mythril before	Mythril after	Slither before	Slither after
1	<code>_0x</code>	1	1	0	0
2	<code>_0xEthereumToken</code>	1	1	0	0
3	AFX	2	2	0	0
4	DEXT	0	0	0	0
5	G4D	1	1	0	0
6	DOGEX	2	2	0	0
7	MiniEthereum	1	1	0	0
8	TIKTOK	1	1	0	0
9	USDx	1	1	0	0
10	ZEBELLION	2	2	0	0
Total		12	12	0	0

As can be observed in Table 7.2, **guard check** and **tight variable packing** patterns did not have an impact on static security analysis in both tools, Mythril (twelve vulnerabilities before and after) and Slither (zero vulnerabilities before and after).

Mythril detected *reentrancy* vulnerability in the following seven contracts (one time in each): `_0x`, `_0xEthereumToken`, AFX, G4D, DOGEX, TIKTOK and USDx. All these seven vulnerabilities could be fixed by implementing the **checks-effects-interaction** or **mutex** patterns.

In AFX, DOGEX and MiniEthereum contracts, Mythril found the *assert violation* vulnerability (one time in each), that could be corrected by changing the *assert* expression for *require* expression.

In ZEBELLION contract, Mythril detected *integer overflow*³ vulnerability (two times), that could be fixed by implementing **safemath** pattern.

³<https://swcregistry.io/docs/SWC-101/>

7.2.3 Maintenance Patterns

Table 7.3 shows the results.

Table 7.3: Maintenance category - security evaluation.

#	Contract	Mythril before	Mythril after	Slither before	Slither after
1	CreativesRegistry	0	0	0	0
2	FaucetPay	2	2	0	0
3	HashTimeLock	0	0	2	2
4	Jug	0	0	2	2
5	MegaPoker	9	9	0	0
6	PostCert	0	0	0	0
7	Proxy	0	0	1	1
8	RigidBit	1	1	0	0
9	Smartex	0	0	2	2
10	UtilitiesRegistry	0	0	0	0
Total		12	12	7	7

As can be observed in Table 7.3, **contract factory** and **mortal** patterns did not have an impact on static security analysis in both tools, Mythril (twelve vulnerabilities before and after) and Slither (seven vulnerabilities before and after).

Mythril detected the *reentrancy* vulnerability seven times in MegaPoker contract, which could be fixed by implementing the **checks-effects-interaction** or **mutex** patterns.

In FaucetPay and RigidBit contracts, Mythril found the *assert violation* vulnerability (one time in each), that could be corrected by changing the Solidity expression *assert* for *require* expression.

In MegaPoker (two times) and FaucetPay (one time) contracts, Mythril detected the *Denial-of-Service (DoS) with fail call*⁴, which could be fixed by avoiding the combination of multiple calls in a single transaction.

Slither detected the *timestamp*⁵ vulnerability two times in HashTimeLock contract, which could be fixed by avoiding the Solidity instruction *block.timestamp* that can be manipulated by miners.

In the Jug contract, Slither found *reentrancy* and *incorrect-equality* vulnerabilities (one time each). The former could be corrected by **checks-effects-interaction** or **mutex** patterns. The latter could be fixed by changing the equality to comparison type of bigger than or smaller than.

Slither detected the *locked-ether*⁶ vulnerability one time in the Proxy contract, which could be corrected by implementing the **pull payment** pattern.

In the Smartex contract, Slither found the *uninitialized-local*⁷ vulnerability one time, which could be corrected by initializing a local variable.

⁴<https://swcregistry.io/docs/SWC-113/>

⁵<https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp/>

⁶<https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether/>

⁷<https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables/>

7.2.4 Security Patterns

Table 7.4 shows the results.

Table 7.4: Security category - security evaluation.

#	Contract	Mythril before	Mythril after	Slither before	Slither after
1	CampaignBank	0	0	2	1
2	CapitalCoinExtended	2	0	0	0
3	Cat	0	0	2	0
4	DaiJoin	3	0	1	1
5	Jobchain	0	0	2	0
6	KriosToken	5	0	1	1
7	QurasToken	1	0	0	0
8	Treelion	1	0	1	1
9	TZVC	0	0	3	0
10	Vow	0	0	2	0
Total		12	0	14	4

As can be observed in Table 7.4, **mutex** and **checks-effects-interaction** patterns eliminated all twelve vulnerabilities detected by Mythril and ten vulnerabilities in a total of fourteen detected by Slither.

Mythril found the *reentrancy* vulnerability in the following contracts: CapitalCoinExtended (two times), DaiJoin (three times), KriosToken (five times), QurasToken (one time) and Treelion (one time). Slither found the same in CampaignBank (one time), Cat (two times), Jobchain (two times), TZVC (three times) and Vow (two times) contracts. All of them were corrected by **mutex** and **checks-effects-interaction** patterns implementation.

Slither detected the *timestamp* vulnerability on time in CampaignBank contract, which could be fixed by avoiding the Solidity instruction *block.timestamp* that can be manipulated by miners.

In DaiJoin contract, Slither found the *arbitrary-send*⁸ vulnerability one time, which could be corrected by implementing the **ownership** pattern.

Slither detected the *unused-return*⁹ vulnerability on time in KriosToken contract, which could be fixed by ensuring that all the return values of functions are used.

In Proxy contract, Slither detected the *locked-ether* vulnerability one time, which could be corrected by implementing the **pull payment** pattern.

⁸<https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations/>

⁹<https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return/>

7.2.5 All Categories Together

In this subsection, the four categories of design patterns were implemented, through the eight patterns mentioned previously, in ten smart contracts. Table 7.5 shows the results.

Table 7.5: All pattern categories - security evaluation.

#	Contract	Mythril before	Mythril after	Slither before	Slither after
1	AnxToken	0	0	1	0
2	ArdCoin	0	0	1	0
3	COINSTO	0	0	1	0
4	GENEToken	0	0	0	0
5	HarmonyOne	0	0	1	0
6	LivenCoin	0	0	5	5
7	SolaToken	2	0	1	0
8	TrueINR	0	0	1	0
9	WhaleCoin	0	0	1	0
10	XMDToken	0	0	1	0
Total		2	0	13	5

As can be observed in Table 7.5, ten in a total of fifteen vulnerabilities were eliminated in the ten contracts modified. Security patterns **checks-effects-interaction** and **mutex** were responsible for vulnerabilities elimination, and implementation of the other six patterns did not interfere with the ten contract's security.

Mythril found the *reentrancy* vulnerability two times in SolaToken contract. Slither found the same one time in AnxToken, ArdCoin, HarmonyOne, SolaToken, TrueINR, WhaleCoin and XMDToken contracts. All of them were corrected by **mutex** and **checks-effects-interaction** patterns implementation.

Slither detected *erc721-interface*¹⁰ and *locked-ether* vulnerabilities in LivenCoin contract. The former could be fixed by setting suitable return values and value-types for ERC-721¹¹ functions. The latter can be corrected by implementing the **pull payment** pattern.

¹⁰<https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-erc721-interface/>

¹¹<https://cointelegraph.com/explained/non-fungible-tokens-explained/>

7.3 Gas

7.3.1 Authorization Patterns

Table 7.6 shows the results.

Table 7.6: Authorization category - gas evaluation in wei.

#	Contract	Consumption before	Consumption after	Difference
1	Acid	2085895	2385428	299533
2	DCEP	2055296	2361042	305746
3	DrepToken	2035236	2298444	263208
4	GLERC20	2025934	2322335	296401
5	OffChainOrFeedPriceFeed	2117999	2400738	282739
6	ProofOfContributionChain	2059671	2362401	302730
7	RVC	2039745	2335995	296250
8	TokenERC20	2024682	2308399	283717
9	WXB	2089323	2366593	277270
10	YSM	2030516	2344480	313964

In the authorization category, all modified contracts increased gas usage compared to original contracts. All increases were resembling.

7.3.2 Control Patterns

Table 7.7 shows the results.

Table 7.7: Control category - gas evaluation in wei.

#	Contract	Consumption before	Consumption after	Difference
1	_0x	5026144	4998256	-27888
2	_0xEthereumToken	5014676	4965127	-49549
3	AFX	5002806	4960665	-42141
4	DEXT	4993211	4972755	-20456
5	G4D	5047699	5016081	-31618
6	DOGEX	4990553	4971850	-18703
7	MiniEthereum	5008030	4979813	-28217
8	TIKTOK	5051028	5019622	-31406
9	USDx	5050198	5021092	-29106
10	ZEBELLION	5001197	4973747	-27450

In the control category all modified contracts decreased gas usage comparatively to original contracts. AFX and _0xEthereumToken contracts had the biggest decrease of gas usage. **Tight variable packing** pattern had special importance here because its main purpose is gas saving.

7.3.3 Maintenance Patterns

Table 7.8 shows the results.

Table 7.8: Maintenance category - gas evaluation in wei.

#	Contract	Consumption before	Consumption after	Difference
1	CreativesRegistry	3807559	3986532	178973
2	FaucetPay	787094	967590	180496
3	HashTimeLock	4501668	4739492	237824
4	Jug	2255536	2497678	242142
5	MegaPoker	958680	1237344	278664
6	PostCert	1712306	1908470	196164
7	Proxy	560074	741858	181784
8	RigidBit	456132	624586	168454
9	Smartex	4232206	4420078	187872
10	UtilitiesRegistry	3762942	3942715	179773

In the maintenance category all modified contracts increased gas usage comparatively to original contracts. MegaPoker and Jug contracts had the biggest increases.

7.3.4 Security Patterns

Table 7.9 shows the results.

Table 7.9: Security category - gas evaluation in wei.

#	Contract	Consumption before	Consumption after	Difference
1	CampaignBank	3222560	3222560	0
2	CapitalCoinExtended	3293033	3382382	89349
3	Cat	2995710	1280389	-1715321
4	DaiJoin	1684599	1755576	70977
5	Jobchain	7527010	7527010	0
6	KriosToken	3026791	3165571	138780
7	QurasToken	3560643	3681098	120455
8	Treelion	4059390	4180565	121175
9	TZVC	3748598	3633744	-114854
10	Vow	3114943	3114943	0

In the security category five modified contracts increased gas usage, two decreased and three maintained gas usage comparatively to original contracts. The fact **checks-effects-interaction** pattern usually only requires a reordering of the code, contributed to contracts that reduced and maintained gas consumption. Therefore, it is surprising that there is no increase in gas consumption in five contracts since pattern **checks-effects-interaction** was not applied alone, but together with the **mutex** pattern, which implies an insertion of more code that means an increase in gas. This may mean that sometimes the pattern **checks-effects-interaction**, in addition to security, also offers optimization in gas consumption. Nevertheless, according to Zou et al. (2019) Remix IDE as a tool to calculate gas usage "...may not fully reflect the effect of changes at source code.". Unfortunately, there are still no tools to measure the use of gas, which are considered solid and reliable by the Solidity developers community.

7.3.5 All Categories Together

In this subsection, the four categories of design patterns were implemented, through the eight patterns mentioned previously, in ten smart contracts. Table 7.10 shows the results.

Table 7.10: All pattern categories - gas evaluation in wei.

#	Contract	Consumption before	Consumption after	Difference
1	AnxToken	3041216	3012388	-28828
2	ArdCoin	2944224	3078938	134714
3	COINSTO	3040372	3012040	-28332
4	GENEToken	4242244	4777560	535316
5	HarmonyOne	3136682	3009300	-127382
6	LivenCoin	4356922	4608036	251114
7	SolaToken	2848310	2321240	-527070
8	TrueINR	3049652	2908670	-140982
9	WhaleCoin	3040704	2922200	-118504
10	XMDToken	3040304	2907610	-132694

As can be observed in Table 7.10, gas usage decreased in seven contracts. The pattern **tight variable packing** influenced directly these improvements. As previously seen, the pattern **checks-effects-interaction** may also have influenced contract modifications that resulted in a decrease in gas consumption. In the three contracts where there was an increase in gas consumption, the benefits of the two patterns mentioned before did not outweigh the increase in code that the implementation of the remaining six patterns provided.

7.4 Conclusions

The work done up to this chapter allow us to conclude that the implementation of Solidity design patterns usually implies a raise in gas usage by smart contracts (except in gas economic patterns and in **checks-effects-interaction** pattern). Therefore, Solidity developers need to take into account that the utilization of design patterns can affect their smart contracts negatively in terms of gas consumption. On the other hand, economic patterns like **tight variable packing** can contribute to making contracts more gas-efficient (as it was expected in section 3.6). Nevertheless, this type of design patterns is usually based on code optimizations that make contracts reading more difficult.

In addition, security patterns help making smart contracts more robust and reliable while decreasing the probability of successful hacking actions (confirmed as expected in section 3.3). The **checks-effects-interaction** pattern can usually be applied without interfering in or increasing gas usage. Maintenance, control and authorization patterns did not increase vulnerabilities number in the modified contracts.

Finally, static security tools and gas measuring tools are very useful to understand the impact of design patterns in smart contracts. Solidity developers need to consider three main aspects when developing a smart contract: utilization of reusable patterns for faster programming, security and gas efficiency.

Chapter 8

Conclusion

In this chapter the final steps of this dissertation are presented. Firstly, a summary of each chapter of this dissertation is introduced. Secondly, the last task of the third and last objective of this work, which consists of the elaboration of a set of recommendations for Solidity developers based on the results of chapter 6 and chapter 7, is fulfilled. Thirdly, the contributions of this work are explained. Fourthly, several suggestions for future work related to this dissertation are described. Finally, the author develops a personal overview of the knowledge this project provided him with.

8.1 Work Summary

Chapter 1 introduced context, problem, objectives and research methodology of this dissertation. This contributed to the accomplishment of the first and second points of methodology in section 1.3.

Chapter 2 explained the concepts and background information that are needed to better understand the content of this document.

Chapter 3 detailed and exhaustive study of solidity design patterns found in recent literature, which partially contributed to the fulfillment of the third point of methodology in section 1.3.

Chapter 4 justified the value of the solution for the problem identified in this dissertation and presented the argument for the need to discover design patterns for Solidity smart contracts.

Chapter 5 completed the fulfillment of the first point of methodology in section 1.3.

Chapter 6 described how the first and the second objectives of this work were accomplished, which contributed to the fulfillment of the third and fourth points of methodology in section 1.3.

Chapter 7 detailed an evaluation, in terms of security and gas consumption, on selected design patterns, that was conducted through its implementation in some smart contracts, which contributed to the fulfillment of the fifth point of methodology in section 1.3.

Chapter 8 established a set of recommendations (the last task of third objective of this work described in section 5.2) for Solidity developers alongside a personal overview, contributions and suggestions for future work.

8.2 Recommendations

The study and analysis executed in the two previous chapters of this dissertation allowed the definition of the following recommendations destined to Solidity developers:

1. Beginner Solidity developers, after learning the basics of this programming language, should learn about Solidity design patterns, to develop smart contracts more rapidly and securely, with reduced gas costs. Learning about design patterns is very useful in any programming language, but in Solidity it gets even more useful and important due to the financial nature of Ethereum. Here, bad programming practices mean monetary losses, as described in section 1.2;
2. Security design patterns are essential to avoid smart contracts vulnerabilities, in a methodical and reusable way. This was demonstrated in this work not only in the evaluation of security in security patterns (subsection 7.2.4) but also in the appraisal of all pattern categories together (subsection 7.2.5). In both evaluations, the number of vulnerabilities detected by Mythril and Slither decreased after the implementation of this type of design pattern;
3. Security tools such as Mythril and Slither are an enormous help due to their static analysis that should be run before smart contracts deployment on the blockchain. As shown in section 6.3, even deployed smart contracts have many vulnerabilities that are detected by this type of tool. Therefore, Solidity developers should get familiar with static security analysis, to program more secure smart contracts;
4. Gas economic design patterns should be taken into account as often as possible, as a means for smart contracts to have fewer financial costs for users. This was demonstrated in this work not only in the evaluation specifically in gas economic patterns (a subcategory of control patterns in subsection 7.3.2) but also in the appraisal of all pattern categories together (subsection 7.3.5). In the former, gas usage decreased in all tested contracts. In the latter, gas consumption decreased in most tested contracts;
5. Solidity developers should establish a balance between design patterns, security and gas usage, according to the requirements of each smart contract. In a contract that requires high performance, gas economic patterns must be a priority. In another, where security is essential, patterns in this category must be considered of the utmost importance. In situations of large-scale and long-term projects, patterns from all categories (except gas economic type, as described in section 7.4) should be used to make the code simpler and more easily readable;
6. To carry out work similar the one presented in this dissertation, Solidity developers should be more disciplined when identifying the design patterns they use. During this work (in subsection 6.2.2) different names were found for each pattern, that are usually not found in the literature. Table 8.1 summarizes it.

Table 8.1: Patterns names found.

#	Category	Pattern	Other names found
1	Authorization	Access restriction	Restriction access; Embedded permission; Time constraint.
2		Multiple authorization	Multi-signature.
3		Ownership	Authorization; Owned; Ownable.
4	Control	Commit and reveal	Encrypting on-chain data; Hash secret.
5		Guard check	
6		Memory array building	Limit storage.
7		Oracle	Oracize; Oracle contract; Chainlink contract.
8		Poll	Vote; Voting contract.
9		Pull payment	Pull over push; Withdrawal contract.
10		Randomness	Random; Random hash; random query; Rand.
11		Safemath	Math.
12		State machine	
13		String equality comparison	Equals.
14		Tight variable packing	Variables packing.
15		Token	Tokenisation; Token ERC20; Standard token; Token standard.
16	Maintenance	Automatic deprecation	Auto deprecation.
17		Contract composer	
18		Contract factory	Factory contract; Factory.
19		Contract register	Contract registry.
20		Contract relay	Proxy; Proxy delegate; Contract observer.
21		Data segregation	Data contract; Contract data; Eternal storage.
22		Mortal	Termination; Suicidal.
23	Security	Satellite	Contract decorator.
24		Balance limit	Limit balance.
25		Checks-effects-interaction	
26		Emergency stop	Circuit breaker; Pausable.
27		Mutex	Reentrancy guard; NoReentrancy; NoReentrant.
28		Rate limit	Limit Rate.
29		Secure ether transfer	Secure transfer.
30		Speed bump	

All the design pattern identifications shown in Table 8.1 that were found in the manual search (detailed and described in subsection 6.2.2) for the dataset of smart contracts, would make an automated search (like the one done in subsection 6.2.1) more accurate.

8.3 Contributions

The main contributions from this dissertation are:

- In this work, a security analysis was performed to patterns that do not belong to the security category, confirming that their implementation does not cause vulnerabilities in smart contracts. In the literature review, no work was found where something similar had been done;
- The various names and identifications (Table 8.1) that were discovered during the manual search for design patterns in the 318 contracts collected are a contribution to future works similar to this project, or its continuation;
- Sixty-two design patterns were covered in the literature review, of which thirty were selected and confirmed in real contracts collected from Ethereum blockchain. Additionally, four more patterns were found and analyzed. Therefore, this work made it possible to obtain information about sixty-six Solidity design patterns in a single document.

8.4 Limitations and Future Work

The main limitations that occurred in this dissertation are related to lack of time or manpower, to look for Solidity design patterns in more smart contracts, through more interactions of the methodology set in section 1.3. As Peffers et al. (2007) said: "At the end of this activity the researchers can decide whether to iterate back to step three to try to improve the effectiveness of the artifact or to continue on to communication and leave further improvement to subsequent projects.". So, ideally, this project should have had more iterations of the design and development phase, to obtain even more clarifying results in the evaluation and conclusion stages, using the knowledge (e.g. Table 8.1 data) and the experience obtained in each iteration. However, this was not possible because this work was carried out in the context of an individual academic project with limited time.

Research work on Solidity design patterns in the case of this particular project, or IT projects, in general, is never a fully finished product. There are always possible technological updates that can arise at any time which can make this type of projects obsolete very quickly. There are also possible improvements that could be added to this kind of project. Among other possible improvements, the following three stand out:

- A bigger dataset of smart contracts that encompasses all verified smart contracts from Etherscan (around six thousand), for a more comprehensive search for Solidity design patterns. It would require more time and teamwork, instead of an individual project. Similar to the work of Durieux et al. (2019) which was focused on security;
- A continuation of this work (publicly available in a repository¹), in order to overcome the limitations in section 7.1, evaluating modifications in more contracts through all design patterns instead of the only eight patterns selected;
- Similar projects to this dissertation but focused on other programming languages for smart contracts development, as in the case of Ethereum and other blockchains like the ones mentioned in subsection 2.2.7;

¹<https://github.com/jmesmoorish/tdmei/>

- To develop a dapp to validate and test all or most of the design patterns found on this work, not only statically but also dynamically in an Ethereum test net or even in the main net.

8.5 Personnal Overview

The author has chosen Ethereum smart contracts as the theme for his research as a means to acquire technical knowledge about a matter that in the past had already sparked his curiosity in terms of finance and economics.

This project was a challenge at all levels, by the combination of technologies it involves. Although almost all of the used technologies were new to the author of this work, putting all of them together in one project was both challenging and interesting, but it was certainly a very hard job and only possible with a high degree of discipline.

Ethereum is a very interesting and enthusiastic platform, essentially due to its power of disruption. Ethereum-based projects in several areas are growing and many different types of apps can be developed in a decentralized way. By taking advantage of blockchain technology to make the creation of trustless and permissionless dapps possible, Ethereum can help the IT and Finance worlds to become freer and less dependents on giant centralized and dictatorial entities or corporations.

References

- Alchemy, New (2018). <https://medium.com/new-alchemy/a-short-history-of-smart-contract-hacks-on-ethereum-1a30020b5fd>. [Online; accessed 20-February-2020].
- Ammous, Saifedean (2018). *The bitcoin standard: the decentralized alternative to central banking*. John Wiley & Sons.
- Antonopoulos, Andreas M (2014). *Mastering Bitcoin: unlocking digital cryptocurrencies*. O'Reilly Media.
- Antonopoulos, Andreas M and Gavin Wood (2018). *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media.
- Bartoletti, Massimo and Livio Pompianu (2017). "An empirical analysis of smart contracts: platforms, applications, and design patterns". In: *International conference on financial cryptography and data security*. Springer, pp. 494–509.
- BinanceInfo (2020). <https://info.binance.com/>. [Online; accessed 12-June-2020].
- Blogs.wsj.com (2016). <https://blogs.wsj.com/cio/2016/02/02/cio-explainer-what-is-blockchain/>. [Online; accessed 05-January-2020].
- Buterin, Vitalik et al. (2014). "Ethereum: A next-generation smart contract and decentralized application platform". In: *white paper 3*, p. 37.
- Charmaz, Kathy and Linda Liska Belgrave (2007). "Grounded theory". In: *The Blackwell encyclopedia of sociology*.
- Chen, Ting et al. (2017). "Under-optimized smart contracts devour your money". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 442–446.
- CoinMarketCap.com (2020). <https://coinmarketcap.com/>. [Online; accessed 15-February-2020].
- ConsenSys (2018). <https://github.com/ConsenSys/Tokens/tree/master/contracts/eip20/>. [Online; accessed 14-March-2020].
- Cuccuru, Pierluigi (2017). "Beyond bitcoin: an early overview on smart contracts". In: *International Journal of Law and Information Technology* 25.3, pp. 179–195.
- DASP (2020). <https://dasp.co/>. [Online; accessed 20-February-2020].
- Destefanis, Giuseppe et al. (2018). "Smart contracts vulnerabilities: a call for blockchain software engineering?" In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, pp. 19–25.
- Durieux, Thomas et al. (2019). "Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts". In: *arXiv preprint arXiv:1910.10601*.
- Ethdocs.org (2016). <https://ethdocs.org/en/latest/ether.html>. [Online; accessed 15-December-2019].
- Ethereum.org (2019). <https://solidity.readthedocs.io/en/v0.6.2/>. [Online; accessed 15-February-2020].
- Etherscan (2020). <https://etherscan.io/chart/gasprice/>. [Online; accessed 07-June-2020].
- Finance-monthly (2019). <https://www.finance-monthly.com/2019/08/the-top-3-cryptocurrencies-what-makes-them-a-success/>. [Online; accessed 12-April-2020].

- Gamma, Erich et al. (1995). "Elements of Reusable Object-Oriented Software". In: *Design Patterns*. massachusetts: Addison-Wesley Publishing Company.
- Garousi, Vahid, Michael Felderer, and Mika V Mäntylä (2016). "The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature". In: *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, pp. 1–6.
- GNSPS (2020). <https://github.com/GNSPS/solidity-bytes-utils/tree/master/contracts/>. [Online; accessed 15-March-2020].
- Grigg, Ian (2017). "Eos-an introduction". In: *White paper*. <https://whitepaperdatabase.com/eos-whitepaper>.
- Huang, Yongfeng et al. (2019). "Smart contract security: A software lifecycle perspective". In: *IEEE Access* 7, pp. 150184–150202.
- Koen, Peter A (2004). "The fuzzy front end for incremental, platform and breakthrough products and services". In: *PDMA Handbook*, pp. 81–91.
- Lielacher, Alex (2020). <https://www.bitcoinmarketjournal.com/smart-contract-platforms/>. [Online; accessed 31-January-2020].
- Lindgreen, Adam and Finn Wynstra (2005). "Value in business markets: What do we know? Where are we going?" In: *Industrial marketing management* 34.7, pp. 732–748.
- Liu, Yue et al. (2018). "Applying design patterns in smart contracts". In: *International Conference on Blockchain*. Springer, pp. 92–106.
- Luu, Loi et al. (2016). "Making smart contracts smarter". In: *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269.
- Marchesi, Lodovica et al. (2020). "Design Patterns for Gas Optimization in Ethereum". In: *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IW-BOSE)*. IEEE, pp. 9–15.
- Mitra, Rajarshi (2019). <https://blockgeeks.com/guides/smart-contract-platforms-comparison-rsk-vs-ethereum-vs-eos-vs-cardano/>. [Online; accessed 31-January-2020].
- Msg-global.com (2019). <https://www.msg-global.com/blog-item/blockchain-moving-beyond-bitcoin/>. [Online; accessed 05-January-2020].
- Nakamoto, Satoshi (2008). "Bitcoin: A peer-to-peer electronic cash system". In: *white paper*.
- Nicola, Susana, Eduarda Pinto Ferreira, and JJ Pinto Ferreira (2012). "A novel framework for modeling value for the customer, an essay on negotiation". In: *International Journal of Information Technology & Decision Making* 11.03, pp. 661–703.
- OpenZeppelin (2020). <https://github.com/OpenZeppelin/openzeppelin-contracts/tree/master/contracts/>. [Online; accessed 10-March-2020].
- Optimum-web.com (2019). <https://www.optimum-web.com/what-is-ethereum-smart-contract/>. [Online; accessed 16-December-2019].
- Peffer, Ken et al. (2007). "A design science research methodology for information systems research". In: *Journal of management information systems* 24.3, pp. 45–77.
- Porru, Simone et al. (2017). "Blockchain-oriented software engineering: challenges and new directions". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, pp. 169–171.
- Rosic, Ameer (2018a). <https://blockgeeks.com/guides/neo-blockchain/>. [Online; accessed 03-January-2020].
- (2018b). <https://blockgeeks.com/guides/what-is-cardano/>. [Online; accessed 05-January-2020].
- Saaty, Thomas L (2008). "Decision making with the analytic hierarchy process". In: *International journal of services sciences* 1.1, pp. 83–98.

- Sampaio, Alberto (2015). "Improving systematic mapping reviews". In: *ACM SIGSOFT Software Engineering Notes* 40.6, pp. 1–8.
- Soren, Rebecca (2020). <https://github.com/rebeccasoren/PollSolidity/>. [Online; accessed 05-March-2020].
- Sqlindia.com (2019). <https://www.sqlindia.com/blockchainat-a-glance/>. [Online; accessed 05-January-2020].
- Stellar.org (2018). <https://www.stellar.org/case-studies/ibm-blockchain-world-wire/>. [Online; accessed 03-January-2020].
- Szabo, Nick (1997). "Formalizing and securing relationships on public networks". In: *First Monday* 2.9.
- Tapscott, Alex and Don Tapscott (2017). "How blockchain is changing finance". In: *Harvard Business Review* 1.9, pp. 2–5.
- Tapscott, Don and Alex Tapscott (2018). *Blockchain revolution: how the technology behind bitcoin and other cryptocurrencies is changing the world*. Portfolio.
- Torgerson, Carole (2003). *Systematic reviews*. Bloomsbury Publishing.
- Volland, Franz (2019). <https://github.com/fravoll/solidity-patterns/>. [Online; accessed 01-March-2020].
- Wessling, Florian et al. (2018). "How much blockchain do you need? towards a concept for building hybrid dapp architectures". In: *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, pp. 44–47.
- Weusecoins.com (2018). <https://www.weusecoins.com/en/questions/>. [Online; accessed 05-January-2020].
- Wohrer, Maximilian and Uwe Zdun (2018a). "Design patterns for smart contracts in the ethereum ecosystem". In: *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, pp. 1513–1520.
- (2018b). "Smart contracts: security patterns in the ethereum ecosystem and solidity". In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, pp. 2–8.
- (2020). "From Domain Specific Language to Code: Smart Contracts and the Application of Design Patterns". In: *IEEE Software*.
- Woodall, Tony (2003). "Conceptualising 'value for the customer': an attributional, structural and dispositional analysis". In: *Academy of marketing science review* 12.1, pp. 1–42.
- Xu, Xiwei et al. (2018). "A pattern collection for blockchain-based applications". In: *Proceedings of the 23rd European Conference on Pattern Languages of Programs*. ACM, p. 3.
- Zeithaml, Valarie A (1988). "Consumer perceptions of price, quality, and value: a means-end model and synthesis of evidence". In: *Journal of marketing* 52.3, pp. 2–22.
- Zou, Weiqin et al. (2019). "Smart contract development: Challenges and opportunities". In: *IEEE Transactions on Software Engineering*.

Appendix A

Authorization Patterns

A.1 Access restriction

A.1.1 Explanation

Sometimes, functions before being executed in Solidity smart contracts, a set of requirements must be met first. These requirements can be applied in different functions of a contract. So this pattern allows programmers to create modifiers used to check requirements that can be reused several times in the same contract.

A.1.2 Related patterns

Guard check.

A.1.3 Sample Code

```

1 pragma solidity ^0.5.0;
2 import "../Ownership.sol";
3
4 contract AccessRestriction is Owned {
5     uint public creationTime = now;
6
7     modifier onlyBefore(uint _time) {require(now < _time);_;}
8
9     modifier onlyAfter(uint _time) {require(now > _time);_;}
10
11     modifier onlyBy(address account) {require(msg.sender == account);_;}
12
13     modifier condition(bool _condition) {require(_condition);_;}
14
15     modifier minAmount(uint _amount) {require(msg.value >= _amount);_;}
16
17     function f() payable onlyAfter(creationTime + 1 minutes) onlyBy(
18         owner) minAmount(2 ether) condition(msg.sender.balance >= 50 ether)
19     {
20         // some code
21     }
22 }
```

Listing A.1: Pattern: Access restriction. Adapted from Wohrer and Zdun (2018a).

A.2 Multiple authorization

A.2.1 Explanation

In the world of dapps it is natural that the execution of some tasks does not rely only on a unique entity (address), but in several participants eliminating single points of failure in a decentralized way. This pattern allows that contract methods can only be executed if a group of participants authorizes them.

A.2.2 Related patterns

Ownership, Roles.

A.2.3 Sample Code

```

1 pragma solidity ^0.5.0;
2
3 contract MultipleAuthorization {
4
5     uint public nonce = 0;
6     uint public threshold;
7     mapping (address => bool) isOwner;
8     address[] public ownersArr;
9
10    constructor (uint threshold_ , address[] owners_) {
11
12        if (owners_.length > 10 || threshold_ > owners_.length || threshold_
13            == 0){
14            throw;
15        }
16
17        for (uint i=0; i<owners_.length; i++) {
18            isOwner[owners_[i]] = true;
19        }
20        ownersArr = owners_;
21        threshold = threshold_;
22    }
23
24    //address recovered from signatures must be strictly increasing
25    function execute(uint8[] sigV, bytes32[] sigR, bytes32[] sigS, address
26        destination , uint value , bytes data) {
27
28        if (sigR.length != threshold){
29            throw;
30        }
31
32        if (sigR.length != sigS.length || sigR.length != sigV.length){
33            throw;
34        }
35
36        bytes32 txHash = sha3(byte(0x19), byte(0), this, destination , value ,
37            data , nonce);
38        address lastAdd = address(0); // cannot have address(0) as an owner
39
40        for (uint i = 0; i < threshold; i++) {
41            address recovered = ecrecover(txHash , sigV[i] , sigR[i] , sigS[i])
42            ;
43        }
44    }
45 }
```

```
39         if (recovered <= lastAdd || !isOwner[recovered]) {  
40             throw;  
41         }  
42         lastAdd = recovered;  
43     }  
44  
45     //all signatures are accounted for  
46     nonce = nonce + 1;  
47     if (!destination.call.value(value)(data)) {  
48         throw;  
49     }  
50 }  
51 }
```

Listing A.2: Pattern: Multiple authorization. Adapted from Liu et al. (2018).

A.3 Ownership

A.3.1 Explanation

Any Ethereum contract address can call functions from other contracts if there are no restrictions to impede that. Sometimes it makes sense that determined methods in a contract can only be called by the address of the contract owner. Through this pattern that problem can be solved in a reusable way, every time some sensitive task should be executed by the contract owner.

A.3.2 Related patterns

Multiple authorization, Roles.

A.3.3 Sample Code

```
1 pragma solidity ^0.5.0;
2
3 contract Ownership {
4     address public owner;
5     event LogOwnershipTransferred(address indexed previousOwner, address
        indexed newOwner);
6
7     modifier onlyOwner() {
8         require(msg.sender == owner);
9     }
10
11     function Owned() public {
12         owner = msg.sender;
13     }
14
15     function transferOwnership(address newOwner) public onlyOwner {
16         require(newOwner != address(0));
17         emit LogOwnershipTransferred(owner, newOwner);
18         owner = newOwner;
19     }
20 }
21 }
```

Listing A.3: Pattern: Ownership. Adapted from Wohrer and Zdun (2018a).

A.4 Roles

A.4.1 Explanation

Similarly to centralized apps, certain dapps require that different types of users (addresses) have different roles in it. This pattern allows the creation of reusable methods to add and remove users from a specific role, as well as to check if some user has a certain role or not.

A.4.2 Related patterns

Ownership, Multiple authorization.

A.4.3 Sample Code

```
1 library Roles {
2
3     struct Role {
4         mapping (address => bool) bearer;
5     }
6
7     function add(Role storage role , address account) internal {
8         require(!has(role , account), "Roles: account already has role");
9         role.bearer[account] = true;
10    }
11
12    function remove(Role storage role , address account) internal {
13        require(has(role , account), "Roles: account does not have role");
14        ;
15        role.bearer[account] = false;
16    }
17
18    function has(Role storage role , address account) internal view
19    returns (bool) {
20        require(account != address(0), "Roles: account is the zero
21        address");
22        return role.bearer[account];
23    }
24 }
```

Listing A.4: Pattern: Roles. Adapted from OpenZeppelin (2020).

Appendix B

Security Patterns

B.1 Balance Limit

B.1.1 Explanation

Financial matters are essential in Ethereum smart contracts due to their nature and original design to manage valuable transactions. So this situation should always be considered by Solidity programmers. To help in this feature the **balance limit** pattern allows the definition of the threshold of a monetary amount to manage and validate operations that cannot exceed a predetermined limit.

B.1.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract BalanceLimit {
4     uint256 public limit;
5
6     function LimitBalance(uint256 value) public {
7         limit = value;
8     }
9
10    modifier limitedPayable() {
11        require(this.balance <= limit);
12    };
13 }
14
15 function deposit() public payable limitedPayable {
16     // some code
17 }
18
19 }
```

Listing B.1: Pattern: Balance limit. Adapted from Wohrer and Zdun (2018b).

B.2 Checks-effects-interaction

B.2.1 Explanation

In Ethereum, contracts can call other contracts to execute their functions (external calls). This feature is the baseline for reentrancy¹ attack exploitation, in which a called contract maliciously modifies the current internal state of the caller contract. To avoid reentrancy, this pattern offers a systematic code instructions ordination, every time external calls are needed. Firstly, the caller contract makes validations (checks), secondly makes all necessary internal state changes (effects) and finally makes the external call (interaction).

B.2.2 Related patterns

Mutex, Secure ether transfer, Pull payment.

B.2.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract checksEffectsInteraction {
4
5     bool public ended = false;
6
7     function auctionEnd() public {
8         // 1. Checks
9         require(now >= auctionEnd);
10        require(!ended);
11        // 2. Effects
12        ended = true;
13        // 3. Interaction
14        beneficiary.transfer(highestBid);
15    }
16
17 }
```

Listing B.2: Pattern: Checks-effects-interaction. Adapted from Wohrer and Zdun (2018b).

¹<https://swcregistry.io/docs/SWC-107>

B.3 Emergency Stop

B.3.1 Explanation

Testing is crucial in Solidity development to avoid bugs and failures in deployed smart contracts, nevertheless, it always can appear. This pattern allows stopping the functioning of a contract in emergency situations, like bugs or security issues discovery in critical tasks or functionalities.

B.3.2 Sample code

```
1 pragma solidity ^0.5.0;
2 import "../authorization/Ownership.sol";
3
4 contract EmergencyStop is Owned {
5     bool public contractStopped = false;
6
7     modifier haltInEmergency {
8         if (!contractStopped) _;
9     }
10
11     modifier enableInEmergency {
12         if (contractStopped) _;
13     }
14
15     function toggleContractStopped() public onlyOwner {
16         contractStopped = !contractStopped;
17     }
18
19     function deposit() public payable haltInEmergency {
20         // some code
21     }
22
23     function withdraw() public view enableInEmergency {
24         // some code
25     }
26 }
```

Listing B.3: Pattern: Emergency stop. Adapted from Wohrer and Zdun (2018b).

B.4 Mutex

B.4.1 Explanation

As **checks-effects-interaction**, this pattern also has as its goal avoid the reentrancy attack, but with a different approach. It uses a boolean variable which works as a locker during external calls. Every time a contract calls an external contract the locker is activated before the call happens and at the end of the call, the locker is deactivated.

B.4.2 Related patterns

Checks-effects-interaction, Secure ether transfer, Pull payment.

B.4.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract Mutex {
4     bool locked = false;
5
6     modifier noReentrancy() {
7         require(!locked);
8         locked = true;
9         _;
10        locked = false;
11    }
12
13    // f is protected by a mutex, thus reentrant calls
14    // from within msg.sender.call cannot call f again
15    function f() noReentrancy public returns (uint) {
16        require(msg.sender.call());
17        return 1;
18    }
19 }
```

Listing B.4: Pattern: Mutex. Adapted from Wohrer and Zdun (2018b).

B.5 Rate Limit

B.5.1 Explanation

Controlling time procedures is very common in Solidity contracts related to gaming or gambling industries, for example. Time control and management is the purpose of this pattern, by regulating the frequency of how many times a specific task can be executed during a predetermined period.

B.5.2 Related patterns

Speed bump.

B.5.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract RateLimit {
4     uint enabledAt = now;
5
6     modifier enabledEvery(uint t) {
7         if (now >= enabledAt) {
8             enabledAt = now + t;
9         }
10    }
11
12    function withdraw() public enabledEvery(uint minutes) {
13        // some code
14    }
15
16 }
17 }
```

Listing B.5: Pattern: Rate limit. Adapted from Wohrer and Zdun (2018b).

B.6 Secure ether transfer

B.6.1 Explanation

Even though sending money is not the only goal of Ethereum blockchain, as it is for Bitcoin, it is a very frequent action and one the most common ways of a contract to make external calls. This pattern intends to standardize the three different ways for securely sending (avoiding reentrancy attack) monetary funds (in the form of ether cryptocurrency) in Solidity smart contracts, namely through these three methods: send, call and transfer.

B.6.2 Related patterns

Mutex, Checks-effects-interaction, Pull payment.

B.6.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract SecureEtherReceiver {
4     function () public payable {}
5 }
6
7 contract SecureEtherSender {
8     SecureEtherReceiver private receiverAdr = new SecureEtherReceiver();
9
10    function sendEther(uint _amount) public payable {
11        if (!address(receiverAdr).send(_amount)) {
12            //handle failed send
13        }
14    }
15
16    function callValueEther(uint _amount) public payable {
17        require(address(receiverAdr).call.value(_amount).gas(35000)());
18    }
19
20    function transferEther(uint _amount) public payable {
21        address(receiverAdr).transfer(_amount);
22    }
23 }
24
25 }
```

Listing B.6: Pattern: Secure ether transfer. Adapted from Volland (2019).

B.7 Speed Bump

B.7.1 Explanation

Automating tasks and eliminating intermediaries are some of the main goals of the Ethereum platform, which is crucial to modernize and accelerate processes. Nevertheless, sometimes it is convenient that some sensitive tasks are executed more slowly. This pattern makes it possible by slowing down the execution of tasks, according to the contract requirements.

B.7.2 Related patterns

Rate limit.

B.7.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract SpeedBump {
4
5     struct Withdrawal {
6         uint amount;
7         uint requestedAt;
8     }
9
10    mapping (address => uint) private balances;
11    mapping (address => Withdrawal) private withdrawals;
12    uint constant WAIT_PERIOD = 7 days;
13
14    function deposit() public payable {
15        if (!(withdrawals[msg.sender].amount > 0))
16            balances[msg.sender] += msg.value;
17    }
18
19    function requestWithdrawal() public {
20        if (balances[msg.sender] > 0) {
21            uint amountToWithdraw = balances[msg.sender];
22            balances[msg.sender] = 0;
23            withdrawals[msg.sender] = Withdrawal({
24                amount: amountToWithdraw,
25                requestedAt: now
26            });
27        }
28    }
29
30    function withdraw() public {
31        if (withdrawals[msg.sender].amount > 0 && now > withdrawals[msg.
32            sender].requestedAt + WAIT_PERIOD){
33            uint amount = withdrawals[msg.sender].amount;
34            withdrawals[msg.sender].amount = 0;
35            msg.sender.transfer(amount);
36        }
37    }
```

Listing B.7: Pattern: Speed bump. Adapted from Wohrer and Zdun (2018b).

Appendix C

Control Patterns

C.1 Commit and reveal

C.1.1 Explanation

In Ethereum, transactions are publicly visible for anyone, but in some cases, it is required that certain contract procedures are treated with confidentiality. This pattern provides a mechanism that encrypts (commit) data temporarily that can be decrypted (reveal) and checked later.

C.1.2 Sample code

```

1 pragma solidity ^0.5.0;
2
3 contract CommitReveal {
4
5     struct Commit {string choice; string secret; stringstatus;}
6     mapping(address => mapping(bytes32 => Commit)) public userCommits;
7
8     event LogCommit(bytes32 , address);
9     event LogReveal(bytes32 , address , string , string);
10
11     function commit(bytes32 _commit) public returns (bool success) {
12         var userCommit = userCommits[msg.sender][_commit];
13         if(bytes(userCommit.status).length != 0) {
14             return false;
15         }
16         userCommit.status = "c";LogCommit(_commit, msg.sender);
17         return true;
18     }
19
20     function reveal(string _choice, string _secret, bytes32 _commit)
21     public returns (bool success) {
22
23         var userCommit = userCommits[msg.sender][_commit];
24         bytes memory bytesStatus = bytes(userCommit.status);
25
26         if(bytesStatus.length == 0) {
27             return false;
28         }else if (bytesStatus[0] == "r") {
29             return false;
30         }
31
32         if (_commit != keccak256(_choice, _secret)) {
33             return false;

```



```
33     }
34
35     userCommit.choice = _choice; userCommit.secret = _secret;
36     userCommit.status = "r";
37     LogReveal(_commit, msg.sender, _choice, _secret);
38     return true;
39 }
40
41 function traceCommit(address _address, bytes32 _commit) public view
42 returns (string choice, string secret, string status) {
43     var userCommit = userCommits[_address][_commit]; require(bytes(
44     userCommit.status)[0] == "r");
45     return (userCommit.choice, userCommit.secret, userCommit.status)
46 ;
47 }
```

Listing C.1: Pattern: Commit and reveal. Adapted from Wohrer and Zdun 2018a.

C.2 Guard check

C.2.1 Explanation

Validations are a very common procedure for Solidity smart contracts. This pattern is usually implemented to validate user inputs, but can also be used to check contract current state, check data returned from other contracts, or to discard conditions that cannot be logically accepted.

C.2.2 Related patterns

Access restriction.

C.2.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract GuardCheck {
4
5     function donate(address addr) payable public {
6         require(addr != address(0));
7         require(msg.value != 0);
8
9         uint balanceBeforeTransfer = this.balance;
10        uint transferAmount;
11
12        if (addr.balance == 0) {
13            transferAmount = msg.value;
14        } else if (addr.balance < msg.sender.balance) {
15            transferAmount = msg.value / 2;
16        } else {
17            revert();
18        }
19
20        addr.transfer(transferAmount);
21        assert(this.balance == balanceBeforeTransfer - transferAmount);
22    }
23 }
```

Listing C.2: Pattern: Guard check. Adapted from Volland (2019).

C.3 Memory array building

C.3.1 Explanation

Gas consumption should be a concern of Solidity programmers, in order to prevent gas waste in smart contracts. This pattern makes the saving of gas possible by suggesting to build an array with public visibility, that should be used to set and get data of the contract, instead of setting and getting data directly from the state variables (persistent memory) of the contract.

C.3.2 Related patterns

Tight variable packing and String equality comparison.

C.3.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract MemoryArrayBuilding {
4
5     struct Item {
6         string name;
7         string category;
8         address owner;
9         uint32 zipcode;
10        uint32 price;
11    }
12
13    Item[] public items;
14
15    mapping(address => uint) public ownerItemCount;
16
17    function getItemIDsByOwner(address _owner) public view returns (uint
18    []) {
19        uint[] memory result = new uint[](ownerItemCount[_owner]);
20        uint counter = 0;
21
22        for (uint i = 0; i < items.length; i++) {
23            if (items[i].owner == _owner) {
24                result[counter] = i;
25                counter++;
26            }
27        }
28        return result;
29    }
```

Listing C.3: Pattern: Memory array building. Adapted from Volland (2019).

C.4 Oracle

C.4.1 Explanation

In some smart contracts, off-chain data is needed for various situations, namely, financial or betting contracts. This pattern creates an oracle contract that acts as a bridge between the blockchain and the external world, supplying trusted data for other contracts.

C.4.2 Sample code

```

1 pragma solidity ^0.5.0;
2
3 contract Oracle {
4     address knownSource = 0x123...; // known source
5     struct Request {bytes data;function(bytes memory) external callback
6     };
7     Request[] requests;
8     event NewRequest(uint);
9
10    modifier onlyBy(address account) {
11        require(msg.sender == account);
12    } _;
13
14    function query(bytes data, function(bytes memory) external callback)
15    public {
16        requests.push(Request(data, callback));
17        NewRequest(requests.length - 1);}
18
19    // invoked by outside world
20    function reply(uint requestID, bytes response) public onlyBy(
21    knownSource) {
22        requests[requestID].callback(response);}
23
24    }
25
26    contract OracleConsumer {
27        Oracle oracle = Oracle(0x123...); // known contract
28
29        modifier onlyBy(address account) {
30            require(msg.sender == account);
31        } _;
32
33        function updateExchangeRate() {
34            oracle.query("USD", this.oracleResponse);
35        }
36
37        function oracleResponse(bytes response) onlyBy(oracle) {
38            // use the data
39        }
40    }

```

Listing C.4: Pattern: Oracle. Adapted from Wohrer and Zdun (2018a).

C.5 Poll

C.5.1 Explanation

Voting systems are a possible implementation of smart contracts that benefit the decentralized nature of blockchain-based platforms like Ethereum. This pattern simplifies this process by creating a set of suited methods and mechanisms to the development of contracts to manage poll systems.

C.5.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract Poll{
4
5     struct Question{
6         string description;
7         bool complete;
8         uint yes;
9         uint no;
10        mapping(address => bool) voters;
11        string answer;
12    }
13
14    address public manager;
15    Question[] public questions;
16
17    modifier restricted(){
18        require(msg.sender == manager);
19        _;
20    }
21
22    constructor() public {
23        manager = msg.sender;
24    }
25
26    function askQuestion(string description) public restricted {
27        Question memory newQuestion = Question({
28            description: description ,complete: false ,
29            yes: 0,no: 0,answer: ''});
30        questions.push(newQuestion);
31    }
32
33    function voteyes(uint index) public{
34        Question storage question = questions[index];
35        require(!question.complete);
36        require(!question.voters[msg.sender]);
37        question.voters[msg.sender] = true;question.yes++;
38    }
39
40    function voteno(uint index) public{
41        Question storage question = questions[index];
42        require(!question.complete);
43        require(!question.voters[msg.sender]);
44        question.voters[msg.sender] = true;question.no++;
45    }
46
47    function updateAnswer(uint index) public restricted{
```

```
48     Question storage question = questions[index];
49     require(!question.complete);
50     if(question.yes >= question.no){
51         question.answer = 'yes';
52     }else{
53         question.answer = 'no';
54     }
55     question.complete = true;
56 }
57
58 function getAnswer(uint index) public restricted view returns (
59 string){
60     Question storage question = questions[index];
61     require(question.complete);
62     return question.answer;
63 }
64
65 function getQuestionLength() public view returns (uint){
66     return questions.length;
67 }
```

Listing C.5: Pattern: Poll. Adapted from Soren (2020).

C.6 Pull payment

C.6.1 Explanation

When a smart contract calls an external contract to make a payment, there is always the possibility for this operation not to succeed, due to several causes, as an example, the receiving contract throws an error. This pattern solves this problem by isolating each external call and swapping the risk of unsuccessful transference of monetary funds (in ether cryptocurrency), from the contract to the user.

C.6.2 Related patterns

Checks-effects-interaction, Mutex, Secure ether transfer.

C.6.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract PullPayment_auction {
4
5     address public highestBidder;
6     uint highestBid;
7     mapping(address => uint) refunds;
8
9     function bid() public payable {
10         require(msg.value >= highestBid);
11         if (highestBidder != 0) {
12             // record the underlying bid to be refund
13             refunds[highestBidder] += highestBid;
14         }
15         highestBidder = msg.sender;
16         highestBid = msg.value;
17     }
18     //pull payment functionality
19     function withdrawRefund() public {
20         uint refund = refunds[msg.sender];
21         refunds[msg.sender] = 0;
22         msg.sender.transfer(refund);
23     }
24 }
```

Listing C.6: Pattern: Pull payment. Adapted from Wohrer and Zdun (2018a).

C.7 Randomness

C.7.1 Explanation

The process of creating random values in the Ethereum platform can be very useful in several smart contract types such as gambling or lottery. This pattern allows the creation of random values in a secure and reusable way, by resorting to oracle contracts that are fed by off-chain mechanisms.

C.7.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract Randomness {
4
5     bytes32 sealedSeed;
6     bool seedSet = false;
7     bool betsClosed = false;
8     uint storedBlockNumber;
9     address trustedParty = 0xCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF;
10
11     function setSealedSeed(bytes32 _sealedSeed) public {
12         require(!seedSet);
13         require(msg.sender == trustedParty);
14         betsClosed = true;
15         sealedSeed = _sealedSeed;
16         storedBlockNumber = block.number + 1;
17         seedSet = true;
18     }
19
20     function bet() public {
21         require(!betsClosed);
22     }
23
24     function reveal(bytes32 _seed) public {
25         require(seedSet);
26         require(betsClosed);
27         require(storedBlockNumber < block.number);
28         require(keccak256(msg.sender, _seed) == sealedSeed);
29         uint random = uint(keccak256(_seed, blockhash(storedBlockNumber
30     ));
31         // Insert logic for usage of random number here
32         seedSet = false;
33         betsClosed = false;
34     }
35 }
```

Listing C.7: Pattern: Randomness. Adapted from Volland (2019).

C.8 Safemath

C.8.1 Explanation

Arithmetical operations are very common on Solidity smart contracts due to Ethereum vocation to make transactions of accounting digital goods. This pattern standardizes the four most frequent math operations (add, subtract, multiply and divide) with their typical validations.

C.8.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 library SafeMath {
4     function add(uint a, uint b) internal pure returns (uint c) {
5         c = a + b;
6         require(c >= a);
7         return c;
8     }
9
10    function sub(uint a, uint b) internal pure returns (uint c) {
11        require(b <= a);
12        c = a - b;
13        return c;
14    }
15
16    function mul(uint a, uint b) internal pure returns (uint c) {
17        c = a * b;
18        require(a == 0 || c / a == b);
19        return c;
20    }
21
22    function div(uint a, uint b) internal pure returns (uint c) {
23        require(b > 0);
24        c = a / b;
25        return c;
26    }
27 }
```

Listing C.8: Pattern: Safemath. Adapted from OpenZeppelin (2020).

C.9 State machine

C.9.1 Explanation

In certain smart contracts, their logic requires several behavioral stages and transitions, such as in the case of gaming contracts. This pattern solves this problem in a reusable way by applying a state machine to delineate and portray different behaviors in a contract through stages and its transitions.

C.9.2 Sample code

```
1 pragma solidity ^0.5.0;
2 contract StateMachine_depositLock {
3
4     enum Stages {AcceptingDeposits , FreezingDeposits , ReleasingDeposits}
5     Stages public stage = Stages.AcceptingDeposits;
6     uint public creationTime = now;
7     mapping (address => uint) balances;
8
9     modifier atStage(Stages _stage) {require(stage == _stage);_;}
10
11    modifier timedTransitions() {
12        if (stage == Stages.AcceptingDeposits && now >=
13            creationTime + 1 days)
14            nextStage();
15        if (stage == Stages.FreezingDeposits && now >=
16            creationTime + 8 days)
17            nextStage();
18        _;
19    }
20
21    function nextStage() internal {stage = Stages(uint(stage) + 1);}
22
23    function deposit() public payable timedTransitions atStage(Stages.
24        AcceptingDeposits) {balances[msg.sender] += msg.value;}
25
26    function withdraw() public timedTransitions atStage(Stages.
27        ReleasingDeposits) {
28        uint amount = balances[msg.sender];
29        balances[msg.sender] = 0;
30        msg.sender.transfer(amount);
31    }
```

Listing C.9: Pattern: State machine. Adapted from Wohrer and Zdun (2018a).

C.10 String equality comparison

C.10.1 Explanation

Gas consumption should be a concern for Solidity programmers, in order to prevent gas waste in smart contracts. This pattern provides a method that simplifies and quickens the comparison of two strings, therefore allowing the reduction of gas usage in this kind of operation.

C.10.2 Related patterns

Memory array building and Tight variable packing.

C.10.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 function hashCompareWithLengthCheck(string a, string b) internal returns
  (bool) {
4
5     if(bytes(a).length != bytes(b).length) {
6         return false;
7     } else {
8         return keccak256(a) == keccak256(b);
9     }
10 }
```

Listing C.10: Pattern: String equality comparison. Adapted from Volland (2019).

C.11 Tight variable packing

C.11.1 Explanation

Gas consumption should be a concern of Solidity programmers, in order to prevent gas waste in smart contracts. This pattern suggests a strategy to save gas by packing contract state variables in a structure in which each variable should reserve only the strictly necessary memory space.

C.11.2 Related patterns

Memory array building and String equality comparison.

C.11.3 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract TightVariablePacking {
4
5     struct CheapStruct {
6         uint8 a;
7         uint8 b;
8         uint8 c;
9         uint8 d;
10        bytes1 e;
11        bytes1 f;
12        bytes1 g;
13        bytes1 h;
14    }
15
16    CheapStruct example;
17
18    function addCheapStruct() public {
19        CheapStruct memory someStruct = CheapStruct(1,2,3,4,"a","b","c",
20        "d");
21        example = someStruct;
22    }
23 }
```

Listing C.11: Pattern: Tight variable packing. Adapted from Volland (2019).

C.12 Token

C.12.1 Explanation

In Ethereum blockchain, it is common to transact value in the form of tokens which can symbolize various types of digital goods such as cryptocurrencies, tickets, or gaming points. This pattern streamlines this process by making standardization of methods which are required for tokens management.

C.12.2 Sample code

```

1 pragma solidity ^0.5.0; import "./EIP20Interface.sol";
2
3 contract Token is EIP20Interface {
4
5     uint256 constant private MAX_UINT256 = 2**256 - 1;
6     mapping (address => uint256) public balances;
7     mapping (address => mapping (address => uint256)) public allowed;
8     string public name, symbol; uint8 public decimals;
9
10    constructor (uint256 _initialAmount, string _tokenName, uint8
11    _decimalUnits, string _tokenSymbol) public {
12        balances[msg.sender] = _initialAmount;
13        totalSupply = _initialAmount;
14        name = _tokenName;
15        decimals = _decimalUnits;
16        symbol = _tokenSymbol;
17    }
18
19    function transfer(address _to, uint256 _value) public returns (bool
20    success) {
21        require(balances[msg.sender] >= _value);
22        balances[msg.sender] -= _value; balances[_to] += _value;
23        emit Transfer(msg.sender, _to, _value);
24        return true;
25    }
26
27    function transferFrom(address _from, address _to, uint256 _value)
28    public returns (bool success) {
29        uint256 allowance = allowed[_from][msg.sender];
30        require(balances[_from] >= _value && allowance >= _value);
31        balances[_to] += _value; balances[_from] -= _value;
32        if (allowance < MAX_UINT256) {
33            allowed[_from][msg.sender] -= _value;
34        }
35        emit Transfer(_from, _to, _value);
36        return true;
37    }
38
39    function balanceOf(address _owner) public view returns (uint256
40    balance) {
41        return balances[_owner];
42    }
43
44    function approve(address _spender, uint256 _value) public returns (
45    bool success) {
46        allowed[msg.sender][_spender] = _value;
47        emit Approval(msg.sender, _spender, _value);
48    }

```

```
43         return true;
44     }
45
46     function allowance(address _owner, address _spender) public view
47     returns (uint256 remaining) {
48         return allowed[_owner][_spender];
49     }
```

Listing C.12: Pattern: Token. Adapted from ConsenSys (2018).

C.13 Address

C.13.1 Explanation

Addresses management is a common operation in Solidity since address is a frequent variable type in smart contracts. This pattern provides Solidity developers with a set of reusable methods to manage address operations. Listing C.13 shows a common method (`isContract`) that checks if an address pertains to a contract account (see subsection 2.2.2).

C.13.2 Sample code

```
1 library Address {
2
3     function isContract(address account) internal view returns (bool) {
4         uint256 size;
5         assembly { size := extcodesize(account) }
6         return size > 0;
7     }
8 }
```

Listing C.13: Pattern: Address. Adapted from OpenZeppelin (2020).

C.14 Byteslib

C.14.1 Explanation

String management is a common operation in Solidity smart contracts, that can sometimes be laborious. This pattern provides Solidity developers with a generous set of reusable methods to manage string operations like concatenation, slicing and casting. Listing C.14 shows two common methods (toAddress and toUint).

C.14.2 Sample code

```
1 library BytesLib {
2
3   function toAddress(bytes _bytes, uint _start) internal pure returns (
4       address) {
5       require(_bytes.length >= (_start + 20));
6       address tempAddress;
7
8       assembly {
9           tempAddress := div(mload(add(add(_bytes, 0x20), _start)), 0
10              x100000000000000000000000000000000)
11       }
12
13       return tempAddress;
14   }
15
16   function toUint(bytes _bytes, uint _start) internal pure returns (
17       uint256) {
18       require(_bytes.length >= (_start + 32));
19       uint256 tempUint;
20
21       assembly {
22           tempUint := mload(add(add(_bytes, 0x20), _start))
23       }
24
25       return tempUint;
26   }
27 }
```

Listing C.14: Pattern: Byteslib. Adapted from GNSPS (2020).

C.15 Context

C.15.1 Explanation

The need to check the current execution context in Solidity smart contracts is a frequent operation. This pattern makes it available by creating two reusable methods to get the sender (address) and the data of a transaction through the access to the globally available variables `msg.sender` and `msg.data`.

C.15.2 Sample code

```
1 contract Context {  
2  
3     constructor () internal { }  
4  
5     function _msgSender() internal view returns (address payable) {  
6         return msg.sender;  
7     }  
8  
9     function _msgData() internal view returns (bytes memory) {  
10        this;  
11        return msg.data;  
12    }  
13 }
```

Listing C.15: Pattern: Context. Adapted from OpenZeppelin (2020).

Appendix D

Maintenance Patterns

D.1 Automatic Deprecation

D.1.1 Explanation

In Solidity smart contracts programming, there is the need to deprecate some tasks or functionalities. This pattern makes this possible by creating an expiration time and some modifiers to manage and validate contract functions according to their deadline to be deprecated.

D.1.2 Sample code

```

1 pragma solidity ^0.5.0;
2
3 contract AutoDeprecation {
4     uint expires;
5
6     function AutoDeprecate(uint _days) public {expires = now + _days * 1
7         days;}
8
9     function expired() internal view returns (bool) {return now >
10        expires;}
11
12     modifier willDeprecate() {require(!expired()); _;}
13
14     modifier whenDeprecated() {require(expired()); _;}
15
16     function deposit() public payable willDeprecate {
17         // some code
18     }
19
20     function withdraw() public view whenDeprecated {
21         // some code
22     }
23 }
```

Listing D.1: Pattern: Automatic deprecation. Adapted from Wohrer and Zdun (2018a).

D.2 Contract composer

D.2.1 Explanation

In Solidity, complex smart contracts sometimes reach a considerable size in terms of the number of lines of code, which can make it difficult to read and understand. This pattern serves to facilitate this situation by creating modular and auxiliary contracts that make up the main contract.

D.2.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 interface Service{
4     function setupContract(string v1 , string v2 , uint v3);
5 }
6
7 contract ContractOne is Service {
8     string var1;
9     string var2;
10    uint var3;
11
12    function setupContract(string v1 , string v2 , uint v3){
13        var1 = v1;
14        var2 = v2;
15        var3 = v3 ;
16    }
17
18    function getInfo constant returns (string , string , uint){
19        return var1 , var2 , var3;
20    }
21    // some code
22 }
23
24 contract ContractTwo is Service {
25     // some code
26 }
27 contract ContractThree is Service {
28     // some code
29 }
30 contract ContractComposer{
31     ContractOne ContractOneService;
32     ContractTwo ContractTwoService;
33     ContractThree ContractThreeService;
34     // some code
35 }
```

Listing D.2: Pattern: Contract composer. Adapted from Liu et al. (2018).

D.3 Contract factory

D.3.1 Explanation

Usually in Solidity smart contracts programming, there is the need to create and deploy multiple instances of a certain contract. It can be painful if done manually in situations where the creation process is in the order of hundreds or thousands. So this pattern allows the automation of this process by creating a contract that acts like a "factory" producing as many instances as needed.

D.3.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract ContractX {
4     address public creator;
5
6     constructor(address c) public {
7         creator = c;
8     }
9 }
10
11 contract ContractX_Factory {
12     address[] public deployedContractsX;
13
14     function createContractX() public {
15         address newContractX = new ContractX (msg.sender);
16         deployedContractX.push(newContractX);
17     }
18
19     function getDeployedContractsX() public view returns (address[]) {
20         return deployedContractsX;
21     }
22 }
```

Listing D.3: Pattern: Contract factory. Adapted from Xu et al. (2018).

D.4 Contract register

D.4.1 Explanation

In certain environments, participants often need to check what is the latest version of a certain smart contract. This pattern makes it possible in an easy and reusable way by creating a contract that registers its latest version. Participants just need to query the register contract to get the latest version.

D.4.2 Related patterns

Contract relay.

D.4.3 Sample code

```
1 pragma solidity ^0.5.0;
2 import "../authorization/Ownership.sol";
3
4 contract ContractRegister is Owned {
5     address backendContract;
6     address[] previousBackends;
7
8     function Register() public {
9         owner = msg.sender;
10    }
11
12    function changeBackend(address newBackend) public onlyOwner()
13    returns (bool) {
14        if(newBackend != backendContract) {
15            previousBackends.push(backendContract);
16            backendContract = newBackend;
17            return true;
18        }
19        return false;
20    }
21 }
```

Listing D.4: Pattern: Contract register. Adapted from Wohrer and Zdun (2018a).

D.5 Contract relay

In certain environments, participants often need to check what is the latest version of a certain smart contract. This pattern makes it possible in an easy and reusable way by creating a contract that acts like a proxy contract that relays all participant's queries to the latest contract version.

D.5.1 Explanation

Contract register.

D.5.2 Sample code

```
1 pragma solidity ^0.5.0;
2 import "../authorization/Ownership.sol";
3
4 contract ContractRelay is Owned {
5     address public currentVersion;
6
7     constructor (address initAddr) public {
8         currentVersion = initAddr;
9         owner = msg.sender;
10    }
11
12    function changeContract(address newVersion) public onlyOwner() {
13        currentVersion = newVersion;
14    }
15
16    // fallback function
17    function f() public {
18        require(currentVersion.delegatecall(msg.data));
19    }
20 }
```

Listing D.5: Pattern: Contract relay. Adapted from Wohrer and Zdun (2018a).

D.6 Data Segregation

D.6.1 Explanation

When a new version of a smart contract is deployed, data stored in its old version needs to migrate. This pattern allows the creation of a contract only with the goal to store the data to feed the contract that contains the logic. This way, it eliminates the need for data migration when contract version updates occur.

D.6.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract DataStorage {
4     mapping(bytes32 => uint) uintStorage;
5
6     function getUintValue(bytes32 key) public constant returns (uint) {
7         return uintStorage[key];
8     }
9
10    function setUintValue(bytes32 key, uint value) public {
11        uintStorage[key] = value;
12    }
13 }
14
15 contract Logic {
16     DataStorage dataStorage;
17
18     function Logic(address _address) public {
19         dataStorage = DataStorage(_address);
20     }
21
22     function f() public {
23         bytes32 key = keccak256("emergency");
24         dataStorage.setUintValue(key, 911);
25         dataStorage.getUintValue(key);
26     }
27 }
```

Listing D.6: Pattern: Data segregation. Adapted from Wohrer and Zdun (2018a).

D.7 Mortal

D.7.1 Explanation

A smart contract can have a limited lifetime or simply stop being necessary after a certain period. This pattern allows contract destruction in a reusable way by implementing a native Solidity code instruction called `selfdestruct`. For security reasons, this action should be executed by the contract owner.

D.7.2 Sample code

```
1 pragma solidity ^0.5.0;
2 import "../authorization/Ownership.sol";
3
4 contract Mortal is Owned {
5
6     function destroy() public onlyOwner {
7         selfdestruct(owner);
8     }
9
10    function destroyAndSend(address recipient) public onlyOwner {
11        selfdestruct(recipient);
12    }
13 }
```

Listing D.7: Pattern: Mortal. Adapted from Wohrer and Zdun (2018a).

D.8 Satellite

D.8.1 Explanation

The fact that Ethereum smart contracts are immutable complicates the process of adding new functionalities to an already deployed contract. This pattern facilitates it by allowing the creation of contracts that act like "satellites" of the main contract and are responsible for new features, avoiding the need to deploy a new version of the main contract.

D.8.2 Sample code

```
1 pragma solidity ^0.5.0;
2
3 contract Satellite {
4     function calculateVariable() public pure returns (uint){
5         // calculate var
6         return 2 * 3;
7     }
8 }
9
10 contract Base is Owned {
11     uint public variable;
12     address satelliteAddress;
13
14     function setVariable() public onlyOwner {
15         Satellite s = Satellite(satelliteAddress);
16         variable = s.calculateVariable();
17     }
18
19     function updateSatelliteAddress(address _address) public onlyOwner {
20         satelliteAddress = _address;
21     }
22 }
```

Listing D.8: Pattern: Satellite. Adapted from Wohrer and Zdun (2018a).

Appendix E

R and Node.js Scripts

E.1 Text Mining Approach

```

1 pkg <- c("readtext", "quantda", "openxlsx")
2 new.pkg <- pkg[!(pkg %in% installed.packages())]
3 if (length(new.pkg)) {install.packages(new.pkg)}
4
5 #install.packages("openxlsx")
6 suppressPackageStartupMessages(library(quantda))
7 library(readtext)
8 library(openxlsx)
9
10 #loading contracts
11 folder <- "D:/Dropbox/0_ISEP/TDMEI/tdmei/collectedContracts"
12 data <- readtext(paste0(folder, "/*.sol"))
13
14 #Create corpus
15 docs <- corpus(data)
16 summary(docs)
17 writeLines(as.character(docs[1]))
18
19 #remove solidity words
20 solidityWords = c("contract", "function", "library", "returns", "return"
21 , "address", "pragma", "pure", "public", "view", "solidity",
22 "require", "+", "=", "<", ">", "uint", "uint256", "uint8", "internal", "
23 external", "payable", "bool", "memory", "bytes", "event",
24 "modifier", "for", "to", "_to", "from", "_from", "emit", "using", "
25 constructor", "mapping", "is", "msg.sender", "|", "string",
26 "this", "true", "false", "if", "else", "revert", "indexed")
27
28 #Build a Document-Feature Matrix (DFM)
29 docs_dfm <- dfm(docs, tolower = TRUE, stem = FALSE, remove =
30 solidityWords, remove_punct = TRUE, remove_numbers = TRUE)
31
32 #Create dictionary to search for patterns by file / contract
33 myDict <- dictionary(list(acesRestriction = c("access restriction", "
34 restriction access", "embedded permission", "time constraint"),
35 ownership = c("ownership", "authorization"), multisig = c("multiple
36 authorization", "multi-signature"),
37 pullPayment = c("pull payment", "pull over push", "withdrawal contract")
38 , stateMachine = c("state machine"),
39 commit = c("commit and reveal", "encrypting on-chain data", "hash
40 secret"), oracle = c("oracle"),
41 token = c("token", "tokenisation"), randomness = c("randomness"), poll =
42 c("poll"), math = c("safemath", "math"),
43 guardCheck = c("guard check"), string = c("string equality comparison"),

```

```

35 variable = c("tight variable packing", "variables packing"), memory = c("
    memory array building"),
36 mortal = c("mortal", "termination"), automatic = c("automatic
    peprecation"),
37 data = c("data segregation", "data contract", "eternal storage"),
    satellite = c("satellite", "contract decorator"),
38 register = c("contract register", "contract registry"), relay = c("
    contract relay", "contract observer", "proxy", "proxy delegate"),
39 factory = c("factory contract", "contract factory"), composer = c("
    contract composer"),
40 checks = c("checks-effects-interaction"), emergency = c("emergency stop"
    , "circuit breaker", "pausable"),
41 speedBump = c("speed bump"), rateLimit = c("rate limit"), mutex = c("
    mutex", "reentrancy guard"),
42 balanceLimit = c("balance limit"), secureTransf = c("secure ether
    transfer" ) )
43
44 #Look for design patterns
45 spills_DFM <- dfm(docs_dfm, dictionary = myDict)
46
47 #Save results
48 write.xlsx(spills_DFM, "found_patterns.xlsx")

```

Listing E.1: Text mining code in R.

E.2 Get Selected Addresses

```

1  const fs = require('fs'); const neatCsv = require('neat-csv')
2  const ObjectsToCsv = require('objects-to-csv')
3  const moment = require('moment'); const axios = require('axios')
4  const apiKey = 'SD89E5TADQC6UMMT5MMA4ZPH6Z4SPJYRBZ'
5  const startDate = moment("01/03/2020 00:00", "D/M/YYYY hh:mm").unix()
6  const endDate = moment("31/03/2020 23:59", "D/M/YYYY hh:mm").unix()
7  const minTransactions = 99
8  let list = []; let transactions = []
9  let startBlock = '' // 9581791 -> hardcoded block corresponding
    "01/03/2020 00:00"
10 let endBlock = '' // 9782309 -> hardcoded block corresponding
    "31/03/2020 23:59"
11
12 fs.readFile('export-verified-contractaddress-opensource-license.csv',
    async (err, data) => {
13     if (err) {console.error('Error: '+err); return}
14     list = await neatCsv(data)
15 })
16
17 const getBlockNumber = async (unixDate) => {
18     try {return await axios.get('https://api.etherscan.io/api?module=
    block&action=getblocknobytime&timestamp='+unixDate+'&closest=before&
    apikey='+apiKey)
19     }catch (err) {console.error(err)}
20 }
21
22 const getStartBlockNumber = async () => {
23     const res = await getBlockNumber(startDate)
24     startBlock = res.data.result
25 }
26

```

```

27 const getEndBlockNumber = async () => {
28     const res = await getBlockNumber(endDate)
29     endBlock = res.data.result
30 }
31
32 getStartBlockNumber()
33 getEndBlockNumber()
34
35 const loadTransactions = async () => {
36     try {
37         for (let i = 0; i < list.length; i++) {
38             let res = await axios.get('http://api.etherscan.io/api?
module=account&action=txlist&address='+list[i].ContractAddress+'&
startblock='+startBlock+'&endblock='+endBlock+'&sort=asc&apikey='+
apiKey)
39             console.log(i+' - '+res.data.result.length)
40             if (res.data.result.length > minTransactions){
41                 let obj = []
42                 obj[0] = {index: i, name: list[i].ContractName, address:
list[i].ContractAddress, txcount: res.data.result.length}
43                 transactions.push(obj[0])
44                 let csv = new ObjectsToCsv(obj)
45                 await csv.toDisk('selectedAddresses.csv', { append: true
46             })
47         }
48     } catch (err) {
49         console.error('Error: '+err)
50     }
51 }
52
53 setTimeout(() => {loadTransactions()}, 1000)

```

Listing E.2: Get selected addresses code in Node.js.

E.3 Get Selected Contracts

```

1 const fs = require('fs')
2 const neatCsv = require('neat-csv')
3 const axios = require('axios')
4 const apiKey = 'SD89E5TADQC6UMMT5MMA4ZPH6Z4SPJYRBZ'
5 let list = []; let contracts = []
6
7 fs.readFile('selectedAddresses.csv', async (err, data) => {
8     if (err) {console.error('Error: '+err); return}
9     list = await neatCsv(data)
10 })
11
12 const loadContracts = async () => {
13     try {
14         for (let i = 0; i < list.length; i++) {
15             let res = await axios.get('https://api.etherscan.io/api?
module=contract&action=getsourcecode&address='+list[i].address+'&
apikey='+apiKey)
16             contracts[i] = {name: list[i].name, code: res.data.result
[0].SourceCode}
17             fs.writeFileSync('./collectedContracts/'+contracts[i].name+
_+list[i].index+'.sol', contracts[i].code);

```

```

18         console.log(i+' - '+list[i].name)
19     }
20 } catch (err) {console.error('Error: '+err)}
21 }
22
23 setTimeout(() => { loadContracts() },1000)

```

Listing E.3: Get selected contracts code in Node.js.

E.4 Remove Duplicated Contracts

```

1 pkg <- c("digest", "stringr", "filesstrings")
2 new.pkg <- pkg[!(pkg %in% installed.packages())]
3 if (length(new.pkg)) {install.packages(new.pkg)}
4 library(digest)
5 library(stringr)
6 library(filesstrings)
7
8 folder1 <- "D:/Dropbox/0_ISEP/TDMEI/collectSmartContractsFromEtherscan/
9   collectedContractsOriginal"
10 folder2 <- "D:/Dropbox/0_ISEP/TDMEI/collectSmartContractsFromEtherscan/
11   duplicated_contracts"
12 filelist <- dir(folder, pattern = ".sol", recursive=TRUE, all.files =
13   TRUE, full.names=TRUE)
14
15 md5s <- sapply(filelist, digest, file=TRUE, algo="md5", length = 5000)
16 duplicate_files <- split(filelist, md5s)
17
18 #now divide the list into duplicates ( length > 1) and uniques ( length
19   - 1)
20 z <- duplicate_files
21 z2 <- sapply(z,function (x){length(x)>1})
22 z3 <- split(z,z2)
23 dupes <- z3$"TRUE"
24
25 # remove duplicated contracts
26 for (i in 1:length(dupes)){
27   for (j in 1:length(dupes[[i]])){
28     if (j > 1){
29       file.move(dupes[[i]][j], folder2)
30     }
31   }
32 }

```

Listing E.4: Remove duplicated contracts code in R.